# Bounded Abstract Effects: Applications to Security

DARYA MELICHER, School of Computer Science, Carnegie Mellon University
VALERIE ZHAO, University of Chicago
JONATHAN ALDRICH, School of Computer Science, Carnegie Mellon University
ANLUN XU, School of Computer Science, Carnegie Mellon University
ALEX POTANIN, School of Engineering and Computer Science, Victoria University of Wellington

Effect systems have been a subject of active research for nearly four decades, with the most notable practical example being checked exceptions in programming languages such as Java. While many exception systems support abstraction, aggregation, and hierarchy (e.g., via class declaration and subclassing mechanisms), it is rare to see such expressive power in more generic effect systems. We designed an effect system around the idea of protecting system resources and incorporated our effect system into the Wyvern programming language. Similar to type members, a Wyvern object can have effect members that can abstract lower-level effects, allow for aggregation, and have both lower and upper bounds, providing for a granular effect hierarchy. We argue that Wyvern's effects capture the right balance of expressiveness and power from the programming language design perspective. We present a full formalization of our effect-system design, show that it allows reasoning about authority and attenuation. Our approach is evaluated through a security-related case study.

CCS Concepts: • **Software and its engineering** → *Object oriented languages*; • **Security and privacy** → *Software security engineering*;

Additional Key Words and Phrases: Effects, effect system, expressiveness, abstraction, language-based security

**ACM Reference Format:**
Darya Melicher, Valerie Zhao, Jonathan Aldrich, Anlun Xu, and Alex Potanin. 2020. Bounded Abstract Effects: Applications to Security. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2020), 43 pages.

## 1 INTRODUCTION

An effect system can be used to reason about the side effects of code, such as reads and writes to memory, exceptions, and I/O operations. Java's checked exceptions is a simple effect system that has found widespread use, and interest is growing in effect systems for reasoning about security [Turbak and Gifford 2008], memory effects [Lucassen and Gifford 1988], and concurrency [Bocchino et al. 2009; Bračevac et al. 2018; Dolan et al. 2017].

**Requirements for a scalable effect system** Unfortunately, effect systems have not been widely adopted, other than checked exceptions in Java, a feature that is widely viewed as problematic [van Dooren and Steegmans 2005]. The root of the problem is that existing effect systems do not provide adequate support for scaling to programs that are larger and have complex structure. Any adequate solution must support *effect abstraction*, *effect composition*, and *path-dependent effects*.

Abstraction is key to achieving scale in general, and a principal form of abstraction is abstract types [Mitchell and Plotkin 1988], a modern form of which appears as abstract type members in Scala [Odersky and Zenger 2005]. Analogously to type abstraction, we define *effect abstraction* as the ability to define higher-level effects in terms of lower-level effects, and potentially to *hide*

Authors' addresses: Darya Melicher, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA, 15213; Valerie Zhao , University of Chicago; Jonathan Aldrich, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA, 15213, jonathan.aldrich@cs.cmu.edu; Anlun Xu, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA, 15213; Alex Potanin, School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington, New Zealand, Wellington, NZ, 6144.

2020. 0164-0925/2020/1-ART1 $15.00
https://doi.org/

that definition from clients of an abstraction. In order to integrate with modularity mechanisms, and by analogy to type members, we define effects using *effect members* of modules or objects. For example, a `file.Read` effect could abstract a lower-level `system.FFI` effect. Then clients of a file should be able to reason about side effects in terms of file reads and writes, not in terms of the low-level calls that are made to the foreign function interface (FFI). In large-scale systems, abstraction should be *composable*. For example, a database component might abstract `file.Read` further, exposing it as a higher-level `db.Query` effect to clients. Clients of the database should be oblivious to whether `db.Query` is implemented in terms of a `file.Read` effect or a `network.Access` effect (in the case that the backend is a remote database).

*Effect polymorphism* is a form of parametric polymorphism that allows functions or types to be implemented generically for handling computations with different effects [Lucassen and Gifford 1988]. In systems at a larger scale, there are various possible effects, and each program component may cause different effects. With effect polymorphism, we can write general code that handles objects with different effects, thereby reducing the amount of replicated code. In practice, we have found that to make effects work well with modules, it is essential to extend effect polymorphism by assigning bounds to effect parameters. We therefore introduce *bounded abstract effects*, which allows programmers to define upper and lower bounds both on abstract effects and on polymorphic effect parameters.

We also leverage *path-dependent effects*, i.e., effects whose definitions depend on an object. This adds expressiveness; for example, if we have two `File` objects, `x` and `y`, we can distinguish effects on one file from effects on the other: the effects `x.Read` and `y.Read` are distinct. Path-dependent effects are particularly important in the context of modules, where two different modules may implement the same abstract effect in different ways. For example, it may be important to distinguish `db1.Query` from `db2.Query` if `db1` is an interface to a database stored in the local file system whereas `db2` is a database accessed over the network.

**Design of the effect system in Wyvern** This paper presents a novel and scalable effect-system design that supports effect abstraction and composition. The abstraction facility of our effect-system is inspired by type members in languages such as Scala. Just as Scala objects may define type members, in our effect calculus, any object may define one or more *effect members*. An effect member defines a new effect in terms of the lower-level effects that are used to implement it. The set of lower-level effects may be empty in the base case or may include low-level effects that are hard-coded in the system. Type ascription can enable information hiding by concealing the definition of an effect member from the containing object's clients. In addition to completely concealing the definition of an effect, our calculus provides bounded abstraction, which exposes upper or lower bounds of the definition of an effect, while still hiding the definition of it.

Just as Scala's type members can be used to encode parametric polymorphism over types, our effect members double as a way to provide effect polymorphism. Bounded effect polymorphism is also provided in our system, because abstract effect members can be bounded by upper or lower bounds. We follow numerous prior Scala formalisms in including polymorphism via this encoding rather than explicitly; this keeps the formal system simpler without giving up expressive power.

Finally, because effect members are defined on objects, our effects are *generative*, even dynamically. This yields great expressivity: each object created at runtime defines a new effect for each effect member in that object so that, for example, we can separately track effects on different `File` objects, statically distinguishing the effects on one object from the effects on another.

**Evaluation and Security Applications.** A promising area of application for effects is software security. For example, in the setting of mobile code, [Turbak and Gifford 2008] proposed that effects could be used to ensure that any untrusted code we download can only access the system resources

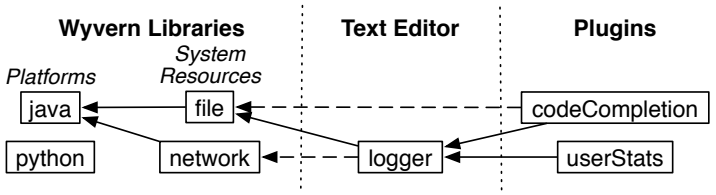


Fig. 1. The overall architecture of the text-editor application. Boxes represent modules, and the arrows represent module imports. The solid arrows are imports that take place, and the dashed arrows represent potential imports that may or may not occur.

it needs to do its tasks, thus following the principle of least privilege [Denning 1976]. We are not aware of prior work that explores this idea in depth.

In order to evaluate our design for effect abstraction, we have incorporated it into an effect system that tracks the use of system resources such as the file system, network, and keyboard. Our effect system is intended to help developers reason about which source code modules use these resources. Through the use of abstraction, we can "lift" low-level resources such as the file system into higher-level resources such as a logging facility or a database and enable application code to reason in terms of effects on those higher-level resources when appropriate. In fact, even the use of resources such as the file system is scaffolded as an abstraction on top of a primitive `system.FFI` effect that our system attaches to uses of the language's foreign function interface. A set of illustrative examples demonstrates the benefits of abstraction for effect aggregation, as well as for information hiding and software evolution. Finally, we show how our effect system allows us to reason about the *authority* [Miller 2006] of code, i.e., what effects a component can have, as well as the *attenuation* of that authority.

Our effect system is implemented in the context of Wyvern, a programming language designed for highly productive development of secure software systems. In this paper, we give several concrete examples of how our effect-system design can be used in software production, all of which are functional Wyvern code that runs in the Wyvern regression test suite.

**Outline and Contributions.** The next section introduces a running example, after which we describe the main contributions of our paper:

- The design of a novel effect system fulfilling the requirements above. Our system is the first to bring together effect abstraction and composition with the effect member construct. Ours is also the first system to provide the programmer with a general form of bounded effect polymorphism and bounded effect abstraction, supporting upper and lower bounds that are other arbitrary effects. (Section 3);
- The application of our effect system to a number of forms of security reasoning, illustrating its expressiveness and making the benefits described above concrete (Section 4);
- A precise, formal description of our effect system, and proof of its soundness. Our formal system shows how to generalize and enrich earlier work on path-dependent effects by leveraging the type theory of DOT (Section 5);
- A formalization of authority using effects, and of authority attenuation (Section 5.7);
- A feasibility demonstration, via the implementation of our approach in the Wyvern programming language (Section 6).

The last sections in the paper discuss related work and conclude.

```
resource type Logger
   effect ReadLog
   effect UpdateLog
   def readLog(): {this.ReadLog} String
   def updateLog(newEntry: String): {this.UpdateLog} Unit

module def logger(f: File): Logger
effect ReadLog = {f.Read}
effect UpdateLog = {f.Append}
def readLog(): {ReadLog} String = f.read()
def updateLog(newEntry: String): {UpdateLog} Unit = f.append(newEntry)
```

Fig. 2. A type and a module implementing the logging facility in the text-editor application.

## 2 RUNNING EXAMPLE

Drawing inspiration from a recent report on security vulnerabilities in text editors [Azouri 2018], we use a text editor application as a running example to demonstrate the key features of our effect system design. The overall architecture of this application is shown in Fig. 1. Each box in the diagram represents a module, and the arrows represent module imports. For the purposes of our forthcoming examples, the solid arrows are imports that take place, and the dashed arrows represent potential imports that may or may not occur.

The application is written using Wyvern's libraries, which contain modules representing system resources, such as the file system and network. These modules rely on access to native backend modules, such as `java` and `python`, which are Wyvern's Java and Python backends, respectively. In the text editor, we focus only on the `logger` module that implements the logging facility of the application. The text editor allows supplementing its core functionality with various third-party plugins. We assume that the application requires that all plugins and user-facing modules of the text editor itself update the log file with the user-observable actions they perform. In our examples, we use two sample plugins: one that, as the user types in code, detects code patterns and offers to complete the code for them, and another that analyzes the text editor's log file and provides insight into how the text editor application is used.

The dashed vertical lines represent the conceptual boundaries between parts of the application that vary in the level of trust based on the security of the contained code. Modules in the Wyvern libraries are the most trusted since they provide functionality essential for all applications developed in Wyvern and were written with security in mind. Modules of the text editor application are less trusted since they are more likely to contain fallible code. Finally, the plugins are the least trusted since they are written by third parties and may be error-prone, vulnerable to exploitation, or outright malicious.

In the following, we will see how Wyvern's effect system can shed light on what effects each module in the systme can have, in terms of the effect abstractions provided by the modules it depends on—ensuring that security vulnerabilities due to modules exceeding their authority are caught by effect checking.

## 3 WYVERN EFFECTS BASICS

Consider the code in Fig. 2 that shows a type and a module implementing the logging facility of the text-editor application. In the given implementation of the `Logger` type, the `logger` module

```
resource type File
   effect Read
   effect Write
   effect Append
   ...
   def read(): {this.Read} String
   def write(s: String): {this.Write} Unit
   def append(s: String): {this.Append} Unit
   ...
```

Fig. 3. The type of the file resource.

accesses the log file.[1] All modules of type `Logger` must have two methods: the `readLog` method that returns the content of the log file and the `updateLog` method that appends new entries to the log file. In addition, the `Logger` type declares two *abstract* effects, `ReadLog` and `UpdateLog`, that are produced by the corresponding methods. These effects are abstract because they are not given a definition in the `Logger` type, and so it is up to the module implementing the `Logger` type to define what they mean. The effect names are user-defined, allowing the choice of meaningful names.

The `logger` module implements the `Logger` type. To access the file system, an object of type `File` (shown in Fig. 3) is passed into `logger` as a parameter. The `logger` module's effect declarations are those of the `Logger` type, except now they are *concrete*, i.e., they have specific definitions. The `ReadLog` effect of the `logger` module is defined to be the `Read` effect of the `File` object, and accordingly, the `readLog` method, which produces the `ReadLog` effect, calls `f`'s `read` method. Similarly, the `UpdateLog` effect of the `logger` module is defined to be `f.Append`, and accordingly, the `updateLog` method, which produces the `UpdateLog` effect, calls `f`'s `append` method. In general, effects in a module or object definition must always be concrete, whereas effects in a type definition may be either abstract or concrete.

### 3.1 Path-dependent Effects

Effects are members of objects[2], so we refer to them with the form `variable.EffectName`, where `variable` is a reference to the object defining the effect and `EffectName` is the name of the effect. For example, in the definition of the `ReadLog` effect of the `logger` module, `f` is the variable referring to a specific file and `Read` is the effect that the `read` method of `f` produces. This conveniently ties together the resource and the effects produced on it (which represent the operations performed on it), helping a software architect or a security analyst to reason about how resources are used by any particular module and its methods. For example, when analyzing the effects produced by `logger`'s `readLog` method, a security analyst can quickly deduce that calling that method affects the file resource and, specifically, the file is read, simply by looking at the `Logger` type and `logger`'s effect definitions but not at the method's code. Furthermore, these properties can be automatically checked with an idiom of use: In addition to directly looking at the effect annotation of the method of the logger module, the security analyst may write client code that specify the effect that the logger module is allowed to have. If the logger module accesses system resources outside of the specified effect set, then the compiler would automatically reject the program.

Because an effect includes a reference to an object instance, our effect system can distinguish reads and writes on different file instances. If the developer does not want this level of precision, it

[1] The keyword `resource` in the type definition indicates that the implementations of this type may have state and may access system resources; this is orthogonal to effect checking.

[2] Modules are an important special case of objects

is still possible to declare effects at the module level (i.e., as members of a `fileSystem` module object instance), and to share the same `Read` and `Write` effects (for example) across all files in `fileSystem`.

The basic mechanisms of path-dependence are borrowed from Scala and have been shown to scale well in practice. These mechanisms come from the Dependent Object Types (DOT) calculus [Amin et al. 2014], a type theory of Scala and related languages (including Wyvern). In our system, effects, instead of types are declared as members of objects.

### 3.2 Effect Abstraction

An important and novel feature of our effect system design is the support for *effect abstraction*. Effect abstraction is the ability to define higher-level effects in terms of lower-level effects and potentially to hide that definition from clients of an abstraction. In the logging example above, through the use of abstraction, we "lifted" low-level resources such as the file system (i.e., the `Read` and `Append` effects of the file) into higher-level resources such as a logging facility (i.e., the `ReadLog` and `UpdateLog` effect of the logger) and enabled application code to reason in terms of effects on those higher-level resources when appropriate.

Effect abstraction has several concrete benefits. First, it can be used to distinguish different uses of a low-level effect. For example, `system.FFI` describes any access to system resources via calls through the foreign function interface (FFI), but modules that define file and network I/O can represent these calls as different effects, which enables higher-level modules to reason about file and network access separately. Second, multiple low-level effects can be aggregated into a single high-level effect to reduce effect specification overhead. For instance, the `db.Query` effect might include both `file.Read` and `network.Access` effects. Third, by keeping an effect abstract, we can hide its implementation from clients, which facilitates software evolution: code defining a high-level effect in terms of lower-level ones can be rewritten (or replaced) to use a different set of lower-level effects without affecting clients (more on this in Section 4.1).

### 3.3 Effect Bounds

Our effect system also gives the programmer the ability to define a subtyping hierarchy of effects via effect bounds. To define the hierarchy, the programmer gives the effect member an upper bound or a lower bound, hiding the definition of the effect from the client.

For example, consider the type `BoundedLogger` which has the same method declarations and effect members as the type `Logger` in Fig. 2, except the `ReadLog` and `UpdateLog` effects are upper-bounded by the corresponding effects in the `fileSystem` module:

```
resource type BoundedLogger
  effect ReadLog <= {fileSystem.Read}
  effect UpdateLog <= {fileSystem.Append}
  ... // same as in the type Logger in Fig. 2
```

Any object implementing type `BoundedLogger` may have an effect member `ReadLog` which is *at most* `fileSystem.Read`. This allows programmers to compare the `ReadLog` effect with other effects, while keeping its definition abstract. For instance, a library can provide two implementations of `BoundedLogger`, including an effectless logger in which the effects `ReadLog` and `UpdateLog` are empty sets, and an effectful logger in which `ReadLog` and `UpdateLog` are defined as effects in the `fileSystem` module. The library's clients then can annotate the effects of both implementations with `fileSystem.Read` and `fileSystem.Append` according to the effect hierarchy, without the need to know the exact implementation of the two instances.

Effect hierarchy can also be constructed using lower bounds. For example, consider the following type for I/O modules that supports writes:

```
type IO
  effect Write >= {system.FFI}
  def write(s: String): {this.Write} Unit
```

Since I/O is done using the foreign function interface (FFI), the `Write` effect is *at least* the `system.FFI` effect. Similar to providing an upper bounded on effects, this type does not specify the exact definition of the `Write` effect, and implementations of this type can define `Write` as an effect set with more effects than `{system.FFI}`.

The effect hierarchy achieved by bounding effect members is supported by the subtyping relations of our effect system (Sections 5.5.1 and 5.5.2). If a type has an effect member with more strict bounds than another type, then the former type is a subtype of the latter type. For example, when a logger with the effect member `Read <= {fileSystem.Read}` is expected, we can pass in a logger with `Read = {}` because the definition as an empty set is more strict than an upper bound.

The following two case studies demonstrates the expressiveness of the effect hierarchy:

*3.3.1 Controlling Access to UI Objects.* This main idea of the work of [Gordon et al. 2013] is to control the access of user interface (UI) framework methods so that unsafe UI methods can only be called by the UI thread. There are three different method annotations `@SafeEffect`, `@UIEffect`, and `@PolyUIEffect`, where

(1) `@SafeEffect` annotates methods that are safe to run on any thread,
(2) `@UIEffect` annotates methods that is only callable on UI thread, and
(3) `@PolyUIEffect` annotates methods whose effect is polymorphic over the receiver type's effect parameter.

In Wyvern, we can model `@UIEffect` as a member of the UI module, for example:

```
type UILibrary
  effect UIEffect >= {system.FFI}
  def unsafeUIMethod1(): {this.UIEffect} Unit
  def unsafeUIMethod2(): {this.UIEffect} Unit
  ...
```

This way, any client code of an UI library that calls UI methods will have the `uilibrary.UIEffect` effect.

An interface could be used for UI-effectful or UI-safe work. To accommodate such flexibility, $Java_{UI}$ introduced the `@PolyUIType` annotation. For example, a `Runnable` interface which can be UI-safe or UI-unsafe is declared as

```
@PolyUIType public interface Runnable {
  @PolyUIEffect void Run();
}
```

Whether the method `Run()` will have a UI effect depends on an annotation when the type is instantiated. For example:

```
@Safe Runnable s =....;
s.run(); // is UI safe
@UI Runnable s = .....;
s.run(); // has UI effect
```

In Wyvern, such polymorphic interface can be created by defining the interface with a bounded effect member:

```
type Runnable
  effect Run <= {uiLibrary.UIEffect}
  def run(): {this.Run} Unit
```

This type ensures that the `run` method is safe to be called on the UI thread. Moreover, if an instance of `Runnable` does not have `UIEffect`, it can be ascribed with the type `SafeRunnable`, which is a subtype of `Runnable`:

```
type SafeRunnable
  effect Run = {}
  def run(): {this.Run} Unit
```

This indicates that `run` is safe to be called on any thread.

*3.3.2 Controlling Mutable States Using Abstract Regions.* [Greenhouse and Boyland 1999] proposed a region-based effect system which describes how state may be accessed during the execution of some program component in object-oriented programming languages. One example of the usage of regions is as follows:

```
class Point {
  public region Position;
  private int x in Position;
  private int y in Position;
  public scale(int sc) reads nothing writes Position {
    x *= sc;
    y *= sc;
  }
}
```

The two variables `x` and `y` are declared inside a region `Position`. For each region, there can be two possible effects: read and write. The `scale` method has the effect of writing on the region `this.Position`.

To achieve access control on regions in Wyvern, we need to keep track of the read and write effect on each variable in a region. We declare the resource type `Var` representing a variable wrapper.

```
resource type Var[T]
  effect Read
  effect Write
  def set (x: T): {this.Write} Unit
  def get (): {this.Read} T
```

Since the `set` and `get` methods are annotated with the corresponding effects and there is no exposed access to the variable that holds the value, the two methods protect the access to the variable inside the type `Var`. To avoid code boilerplate, this wrapper type can be added as a language extension. The `Point` example above can be rewritten in Wyvern as:

```
resource type Point
  val x: Var[Int]
  val y: Var[Int]
  effect Read >= {this.x.Read, this.y.Read}
  effect Write >= {this.x.Write, this.y.Write}
  def scale(sc: Int): {this.Write} Unit
```

We can also extend the type `Point` to `3DPoint` in the following way:

```
resource type 3DPoint
  val x: Var[Int]
  val y: Var[Int]
  val z: Var[Int]
  effect Read = {this.x.Read, this.y.Read, this.z.Read}
  effect Write = {this.x.Write, this.y.Write, this.z.Write}
```

```
def scale(sc: Int): {this.Write} Unit
```

Since the effect `Read` and `Write` in the type `Point` is declared with a lower bound, the type `3DPoint` is a subtype of `Point`.

### 3.4 Effect Aggregation

Wyvern's effect-system design allows reducing the effect-annotation overhead by aggregating several effects into one. For example, if, to update the log file, the `logger` module needed to first read the file and then write it back, the `UpdateLog` effect would consist of two effects: a file read and a file write. In other effect systems, this change may make effects more verbose since all the methods that call the `updateLog` method would need to be annotated with the two effects. However, effect aggregation allows us to define the `UpdateLog` effect to be the two effects and then use `UpdateLog` to annotate the `updateLog` method and all methods that call it:

```
module def logger(f: File): Logger
effect UpdateLog = {f.Read, f.Write}
def updateLog(newEntry: String): {this.UpdateLog} Unit
...
```

This way we need to use only one effect, `UpdateLog`, instead of two, in method effect annotations, thus reducing the effect-annotation overhead. Because more code may add more effects, larger software systems might experience a snowballing of effects, when method annotations have numerous effects in them.

### 3.5 Controlling FFI Effects

Wyvern programs access system resources via calls to other programming languages, such as Java and Python, i.e., through a foreign function interface (FFI). To monitor and control the effects caused by FFI calls, we enforce that all functions from other programming languages, when called within Wyvern, are annotated with the `system.FFI` effect.

As was mentioned in Section 3.2, the `system.FFI` effect is an effect that describes function calls though an FFI. Since every function call though FFI has this effect, the access to system resources via FFI is guaranteed to be monitored. `system.FFI` is the lowest-level effect in the effect system which can be used to build other higher-level effects. The programmer can lift `system.FFI` to higher-level effects and reason about those higher-level effects instead.

For example, Wyvern's import mechanism works by loading an object in a static field of a Java class, and the following code imports a field of a Java class that helps to implement file IO:

```
import java:wyvern.stdlib.support.FileIO.file
```

The file object is itself of type FileIO. And FileIO has this method, among others:

```
public void writeStringIntoFile(String content, String filename) throws IOException { ... }
```

In Wyvern, there is a type wyvern.stdlib.support.FileIO as well as an object file (of that type) that gets added to the scope as a result of the import above. The type has the following member, corresponding to the method above:

```
def writeStringIntoFile(content:String, filename:String): { system.FFI } Unit
```

Here, the system.FFI effect was added to the signature because this is a function that was imported via the FFI. The Wyvern file library that uses the `writeStringIntoFile` function abstracts this `system.FFI` effect into a library-specific `FileIO.Write` effect.

```
module def remoteLogger(net: Network): Logger
effect ReadLog = {net.Receive}
effect UpdateLog = {net.Send}
def readLog(): {ReadLog} String = net.receive()
def updateLog(newEntry: String): {UpdateLog} Unit = net.send(newEntry)
```

Fig. 4. An alternative implementation of the `Logger` type from Fig. 2.

## 4 USE CASES

In this section, we present a selected set of software development patterns that Wyvern's effect system helps facilitate.

### 4.1 Information Hiding and Polymorphism

Introduced by Parnas in the early 1970s [Parnas 1971, 1972], information hiding is a key software development principle stating that, in a software application, implementation details of a particular software module should be hidden behind a stable interface. This principle promotes modularity in the software implementation and gives software developers more flexibility to modify the existing implementation of a module without affecting other modules. Our effect-system design facilitates the principle of information hiding.

For example, Fig. 4 shows an alternative implementation of the `Logger` type from Fig. 2. In this version, the log file is stored on some remote machine, and the network (instead of the file system) is used to perform operations on the log. Importantly, the `Logger` type contains no information about what resource should be used to implement the logging functionality, and thus, a module implementing the `Logger` type may use any resource or no resources at all (in which case `Logger`'s effects could be defined as empty effects, i.e., `{}`). Yet the client modules that use a resource of type `Logger`, such as the two text editor's plugins, observe no difference in the logging functionality. The software architect may swap one `logger` version for the other at any time without affecting the modules using `logger`, provided that the interface of the `Logger` type remains the same. Thus, using effect abstraction in the `Logger` type facilitates the principle of information hiding.

Information hiding is also facilitated by the bounded abstraction feature of our effect system. Consider the following type, which is a subtype of `Logger` and can be ascribed to the `logger` module defined in Fig. 4:

```
type RemoteLogger
  effect ReadLog <= {net.Receive}
  effect UpdateLog <= {net.Send}
  ... // same as in the type Logger in Fig. 2
```

In contrast to `Logger`, `RemoteLogger` defines the `ReadLog` and `UpdateLog` effects as subeffects of `{net.Receive}` and `{net.Send}`, essentially hiding the definitions. Then, if `RemoteLogger` is used as `logger`'s return type, `logger`'s two effects may be defined using the network resource and `logger`'s clients can use the `net`'s effects to account for effects of `logger`'s methods. However, higher-level `logger`'s effects cannot be used to annotate methods that produce lower-level `net`'s effects. Thus, the effect hierarchy allows programmers to annotate methods that have higher-level effects with lower-level effects, but not the other way around.

Our design supports effect polymorphism, as well. For example, the following higher-order function can be used to invoke a function with an arbitrary effect:

```
module def codeCompletion(log: Logger)
def findTemplate(wordSequence: String): {log.UpdateLog} String
   log.updateLog("Searching for a matching template.") ...
   ... log.updateLog("Found matching template.") ...

module def userStats(log: Logger)
def calculateUserStats(): {log.ReadLog, log.UpdateLog} String
   log.updateLog("Starting to analyze the log content.")
   analyzeLogContent(log.readLog()) ...
```

Fig. 5. Excerpts from the code-completion and user-statistics-analyzer plugins of the text-editor application.

```
def invokeTwice[effect E](f: Unit -> {E} Unit)
   f()
   f()
invokeTwice[log.UpdateLog]( () -> log.updateLog("Updating log.") )
```

Here `invokeTwice` is parameterized by an effect `E`. The `invokeTwice` function takes another function that has no arguments and produces no result but has effect `E`, and invokes that function twice. We call `invokeTwice`, instantiate the effect parameter with `log.UpdateLog`, and give `invokeTwice` a function that updates the log file.

Our implementation follows Scala's approach to type polymorphism. Internally, effect polymorphism is rewritten in terms of effect members, so that `invokeTwice` takes an extra argument that has an effect member `E`.

### 4.2 Controlling Operations Performed on Modules

Our effect system design allows software developers to control what operations are performed on system resources and other important modules. Consider the two plugins for the text editor. As we noted earlier, these plugins lie outside the trusted code base for the application because they were written by third parties and may contain bugs, which could introduce vulnerabilities, or be malicious. To better maintain security of the text-editor application and minimize any potential damage from the plugins, developers of the text editor need to control what resources the plugins access and what operations they perform on those resources. The first part of this task, i.e., controlling access to resources, is done via Wyvern's capability-based module system, which limits the plugins' access to resources [Melicher et al. 2017]. The second part of the task, i.e., limiting what operations are performed on the resources the plugins have access to, is where Wyvern's effect system can help.

For example, Fig. 5 shows some code of the two text editor's plugins. Both plugins have access to the `logger` module, which is passed in as a functor parameter, but they use it differently. Both plugins must follow the text editor's policy of recording user-observable actions they perform, but only the `userStats` plugin needs to perform more operations on `logger` than simply updating it. The `codeCompletion` module needs `logger` only to update the log file about the status of the search of an appropriate template in its `findTemplate` method. On the other hand, along with updating the log file, the `userStats` module reads the log file to analyze its content. Accordingly, `codeCompletion`'s `findTemplate` method is annotated with the `log.UpdateLog` effect and therefore must only call `logger`'s `updateLog` method. In contrast, `userStats`'s `calculateUserStats` method is annotated with both the `log.ReadLog` and `log.UpdateLog` effects and may therefore call either `updateLog` or `readLog` on the `logger`.

Wyvern's effect system ensures that the method bodies of `findTemplate` and `calculateUserStats` methods produce only the effects with which the methods are annotated (more details on this are in Section 5). Then, the developer can rely on the effect annotations in modules' interfaces to reason

```
module fileEffects: FileEffects
effect Read = {system.FFI}
effect Write = {system.FFI}
effect Append = {system.FFI}
...
```

(a) A pure module defining file effects.

```
type FileEffects
  effect Read >= {system.FFI}
  effect Write >= {system.FFI}
  effect Append >= {system.FFI}
...
```

(b) A type for the module containing file effects which provides lower bounds for them.

Fig. 6. Defining globally available file effects.

about the effects that methods may produce on resources. Thus, our effect-system design allows controlling what operations are performed on resources of an application and also significantly simplifies the reasoning process during an analysis of the application security.

### 4.3 Effect Granularity and Visibility

Our approach offers library designers a choice of granularity of effects: effects that are defined per-object on the one hand, vs. globally-defined effects shared by many objects on the other hand. For instance, the example code shown so far has envisioned effects for `File` objects that are specific to each individual file, allowing fine-grained control of what code accesses what file. Using fine-grained effects, for example, we can verify that the `logger` accesses a single, distinguished log file, and no other files.

In a different design for the file system libraries, we might define coarse-grained `Read`, `Write`, and `Append` effects in a globally-accessible module. For example, Fig. 6a shows a `fileEffects` module that defines these effects in terms of Wyvern's low-level foreign function access effect, `system.FFI`. We hide this concrete definition behind the `FileEffects` module type defined in Fig. 6b. `FileEffects` puts a lower bound of `system.FFI` on each of these effects, which has two purposes. First, the code implementing file reads and writes, which does so using the foreign function interface and therefore incurs the low-level `system.FFI` effect, can be annotated with higher level effects such as `fileEffects.Read`, since `fileEffects.Read` is known to subsume `system.FFI`. Second, file system client code has to assume that, in general, `fileEffects.Read` might include more than `system.FFI`, since it cannot see the true definition of `Read` in the `fileEffects` module due to the ascribed `FileEffects` signature, which hides this definition. Thus, client code must treat `fileEffects.Read` abstractly; it cannot treat it as merely being `system.FFI` or as any other effect implemented in terms of `system.FFI`.

Thus, Wyvern's design elegantly supports either local, fine-grained definitions of effects, such as reads on a particular file, or more coarse-grained effects, such as reads to any file in the file system. It is even possible to combine these designs; for example, we could define `fileEffects.Read` as shown above, and then type `File` could declare a `Read` effect that is specific to that file, but yet is a sub-effect of `fileEffects.Read`. In this design, a method that takes a file argument `f` and reads from it could be annotated with a fine-grained `f.Read` effect, while a caller of that method that accesses multiple files could declare its effects with the more coarse-grained `fileEffects.Read` to keep its declaration succint.

### 4.4 Authority Attenuation

A key principle to ensure security of a software system is the principle of least privilege [Denning 1976], which states that a software module must only have privilege necessary to implement its designated functionality and nothing else. In practice, this principle translates into protecting software modules representing system resources, such as the file system and network, and other important modules, such as those holding user data, from excessive access and abuse by other

software modules. An important component of privilege is operations performed on a resource being accessed. In the field of software security, such operations represent *authority* over the accessed module [Miller 2006].[3]

Notably, Wyvern effects that describe operations performed on modules are a good medium for representing authority over modules. For example, the fact that the `logger` module's effects use only `file`'s `Read` and `Append` effects in `logger`'s effect definitions signifies that the only operations `logger` performs on the log file are the read and append operations, meaning that the only authority `logger` has over the log file is to read it and append to it.

Furthermore, our effect-system design allows expressing the notion of *authority attenuation*, which is a common software-security pattern [Murray 2008]. Authority attenuation happens when the original set of operations that can be performed on a resource is limited by an intermediary object [Miller 2006]. For example, consider the sequence of module dependencies from Fig. 1 consisting of the `file` module, the `logger` module, and the `codeCompletion` module. There are several operations that can be performed on a file (at least the three shown in the `File` type in Fig. 3), but `logger` performs only two of them (as was mentioned above and as can be seen from its effects' definitions in Fig. 2). The `codeCompletion` module can access the `logger` module but not the `file` module, and so, the only operations it can perform on `file` are those that `logger` can perform. Thus, the `logger` module attenuates `codeCompletion`'s authority over the `file` module.

Therefore, our effect-system design helps developers in observing and establishing the authority attenuating relationship between modules of a software application, which may be desired and beneficial during the design phase of a software application, a security audit, or an architecture review of a software application.

## 5 FORMALIZATION

As was mentioned earlier, Wyvern modules are first class and are, in fact, objects since they are only syntactic sugar on top of Wyvern's object-oriented core and can be translated into objects. The translation has been described in detail previously [Melicher et al. 2017], and here we provide only some intuition behind it. In this section, we start with describing the syntax of Wyvern's object-oriented core, then present an example of the module-to-object translation, followed by a description of Wyvern's static semantics and subtyping rules. Furthermore, we present the dynamic semantics and the type soundness theorems. Last but not least, we provide the definitions on authority and discuss why they are useful for security analysis on programs written in Wyvern.

### 5.1 Object-Oriented Core Syntax

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \mathsf{new}(x \Rightarrow \overline{d}) \\
 & | & e.m(e) \\
 & | & e.f \\
 & | & e.f = e
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & \mathtt{def}\ m(x:\tau):\{\varepsilon\}\ \tau = e \\
 & | & \mathtt{var}\ f:\tau = x \\
 & | & \mathtt{effect}\ g = \{\varepsilon\} \\
\varepsilon & ::= & \overline{x.g} \\
\tau & ::= & \{x \Rightarrow \overline{\sigma}\} \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x:\tau
\end{array}
\qquad
\begin{array}{rcl}
\sigma & ::= & \mathtt{def}\ m(x:\tau):\{\varepsilon\}\ \tau \\
 & | & \mathtt{var}\ f:\tau \\
 & | & \mathtt{effect}\ g \\
 & | & \mathtt{effect}\ g \geqslant \{\varepsilon\} \\
 & | & \mathtt{effect}\ g \leqslant \{\varepsilon\} \\
 & | & \mathtt{effect}\ g = \{\varepsilon\}
\end{array}
$$

Fig. 7. Wyvern's object-oriented core syntax.

Fig. 7 shows the syntax of Wyvern's object-oriented core. Wyvern expressions include variables and the four basic object-oriented expressions: the `new` statement, a method call, a field access, and a field assignment. Objects are created by `new` statements that contain a variable $x$ representing the

[3] Similar to the work by Maffeis et al. [Maffeis et al. 2010], we widened the original definition of authority to be about being able to perform any operation on a module, instead of being able to only modify it.

current object along with a list of declarations. In our implementation, $x$ defaults to `this` when no name is specified by the programmer. Declarations come in three kinds: a method declaration, a field, and an effect member. Method declarations are annotated with a set of effects. Object fields may only be initialized using variables, a restriction which simplifies our core language by ensuring that object initialization never has an effect. Although at first this may seem to be limiting, in fact, we do not limit the source language in this way. Side-effecting member initializations in the source language are translated to the core by wrapping the new object with a `let` expression (a discussion of which is upcoming) that defines the variable to be used in the field initialization. For example, this code:

```
new
  var x: String = f.read()
```

can be internally rewritten as:

```
let y = f.read()
in new
  var x: String = y
```

Effects in method annotations and effect-member definitions are surrounded by curly braces to visually indicate that they are sets, and each effect in an effect set is defined to be a variable representing the object on which an effect is produced, followed by a dot and the effect name. Abstract effects may be defined with an upper bound or a lower bound.

Object types are a collection of declaration types, which include method signatures, field-declaration types, and the types of effect-member declarations and definitions. Similar to the difference between the modules and their types, effects in an object must always be defined (i.e., always be concrete), whereas effects in object types may or may not have definitions (i.e., be either abstract or concrete), and may have an upper or lower bound.

### 5.2 Modules-to-Objects Translation

```
1  let logger = new(x ⇒
2    def apply(f : File) : {} Logger
3        new(_ ⇒
4          effect ReadLog = {f.Read}
5          effect UpdateLog = {f.Append}
6          def readLog() : {ReadLog} String = f.read()
7          def updateLog(newEntry : String) : {UpdateLog} Unit = f.append(newEntry)))
8  in .../// calls logger.apply(...)
```

Fig. 8. A simplified translation of the `logger` module from Fig. 2 into Wyvern's object-oriented core.

Fig. 8 presents a simplified translation of the `logger` module from Fig. 2 into Wyvern's object-oriented core (for a full description of the translation mechanism, refer to [Melicher et al. 2017]). For our purposes, the functor becomes a regular method, called `apply`, that has the return type `Logger` and the same parameters as the module functor. The method's body is a new object containing all the module declarations. The `apply` method is the only method of an outer object that is assigned to a variable whose name is the module's name. Later on in the code, when the `logger` module needs to be instantiated, the `apply` method is called with appropriate arguments passed in.

To aid this translation mechanism, we use the two relatively standard encodings:

$$\texttt{let } x = e \texttt{ in } e' \;\equiv\; \texttt{new}(\_ \Rightarrow \texttt{def } f(x : \tau) : \tau' = e').f(e)$$

$$\texttt{def } m(\overline{x : \tau}) : \tau = e \;\equiv\; \texttt{def } m(x : (\tau_1 \times \tau_2 \times ... \times \tau_n)) : \tau = [x.n/x_n]e$$

The `let` expression is encoded as a method call on an object that contains that method with the `let`

$\boxed{\Gamma \vdash \tau\ wf}$

$$\frac{\forall \sigma \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \vdash \sigma\ wf}{\Gamma \vdash \{x \Rightarrow \overline{\sigma}\}\ wf}\ (\text{WF-Type})$$

$\boxed{\Gamma \vdash \sigma\ wf}$

$$\frac{\Gamma \vdash \tau_2\ wf \quad \Gamma,\ x : \tau_2 \vdash \tau_1\ wf \quad \Gamma,\ x : \tau_2 \vdash \varepsilon\ wf}{\Gamma \vdash \mathsf{def}\ m(x : \tau_2) : \{\varepsilon\}\ \tau_1\ wf}\ (\text{WF-Def}) \qquad \frac{\Gamma \vdash \tau\ wf}{\Gamma \vdash \mathsf{var}\ f : \tau\ wf}\ (\text{WF-Var})$$

$$\frac{}{\Gamma \vdash \mathsf{effect}\ g\ wf}\ (\text{WF-Effect1}) \qquad \frac{\Gamma \vdash \varepsilon\ wf}{\Gamma \vdash \mathsf{effect}\ g = \{\varepsilon\}\ wf}\ (\text{WF-Effect2})$$

$$\frac{\Gamma \vdash \varepsilon\ wf}{\Gamma \vdash \mathsf{effect}\ g \leqslant \{\varepsilon\}\ wf}\ (\text{WF-Effect3}) \qquad \frac{\Gamma \vdash \varepsilon\ wf}{\Gamma \vdash \mathsf{effect}\ g \geqslant \{\varepsilon\}\ wf}\ (\text{WF-Effect4})$$

$\boxed{\Gamma \vdash \varepsilon\ wf}$

$$\frac{\begin{array}{c}\forall i, j,\ x_i.g_j \in \varepsilon,\ \Gamma \vdash safe(x_i.g_j, \{\}),\ \Gamma \vdash x_i : \{\}\ \{y_i \Rightarrow \overline{\sigma_i}\},\\ (\mathsf{effect}\ g_j \in \overline{\sigma_i} \vee \mathsf{effect}\ g_j = \{\varepsilon_j\} \in \overline{\sigma_i} \vee \mathsf{effect}\ g_j \geqslant \{\varepsilon_j\} \in \overline{\sigma_i} \vee \mathsf{effect}\ g_j \leqslant \{\varepsilon_j\} \in \overline{\sigma_i})\end{array}}{\Gamma \vdash \varepsilon\ wf}\ (\text{WF-Effect})$$

$\boxed{\Gamma \vdash safe(x.g, \varepsilon)}$

$$\frac{\begin{array}{l}\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \mathsf{effect}\ g = \{\varepsilon'\} \in \overline{\sigma}\\ \forall a.b \in \{x.g\} \cup \varepsilon, a.b \notin [x/y]\varepsilon'\\ \forall c.d \in [x/y]\varepsilon', \Gamma \vdash safe(c.d, \{x.g\} \cup \varepsilon)\end{array}}{\Gamma \vdash safe(x.g, \varepsilon)}\ (\text{Safe-1})$$

$$\frac{\begin{array}{l}\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \mathsf{effect}\ g \geqslant \{\varepsilon'\} \in \overline{\sigma}\\ \forall a.b \in \{x.g\} \cup \varepsilon, a.b \notin [x/y]\varepsilon'\\ \forall c.d \in [x/y]\varepsilon', \Gamma \vdash safe(c.d, \{x.g\} \cup \varepsilon)\end{array}}{\Gamma \vdash safe(x.g, \varepsilon)}\ (\text{Safe-2})$$

$$\frac{\begin{array}{l}\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \mathsf{effect}\ g \leqslant \{\varepsilon'\} \in \overline{\sigma}\\ \forall a.b \in \{x.g\} \cup \varepsilon, a.b \notin [x/y]\varepsilon'\\ \forall c.d \in [x/y]\varepsilon', \Gamma \vdash safe(c.d, \{x.g\} \cup \varepsilon)\end{array}}{\Gamma \vdash safe(x.g, \varepsilon)}\ (\text{Safe-3})$$

$$\frac{\Gamma \vdash x : \{\}\{y \Rightarrow \overline{\sigma}\},\ \mathsf{effect}\ g}{\Gamma \vdash safe(x.g, \varepsilon)}\ (\text{Safe-4})$$

Fig. 9. Wyvern well-formedness rules.

variable being the method's parameter and the method body being the `let`'s body. The multiparameter version of the method definition is encoded using indexing into the method parameters.

### 5.3 Well-formedness

Since Wyvern's effects are defined in terms of variables, before we type check expressions, we must make sure that effects and types are well formed. Wyvern well-formedness rules are mostly straightforward and are shown in Fig. 9. The three judgements read that, in the variable typing context $\Gamma$, the type $\tau$, the declaration type $\sigma$, and the effect set $\varepsilon$ are well formed, respectively.

An object type is well formed if all of its declaration types are well formed. A method-declaration type is well formed if the type of its parameter, its return type, and the effects in its effect annotation

$\boxed{\Gamma \vdash e : \{\varepsilon\}\ \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \{\}\ \tau}\ \text{(T-Var)} \qquad \frac{\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \vdash d_i : \sigma_i}{\Gamma \vdash \mathsf{new}(x \Rightarrow \overline{d}) : \{\}\ \{x \Rightarrow \overline{\sigma}\}}\ \text{(T-New)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \{\varepsilon_1\}\{x \Rightarrow \overline{\sigma}\} \quad \mathsf{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma} \\ \Gamma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf \quad \Gamma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2 \quad \varepsilon = \varepsilon_1 \cup \varepsilon_2 \cup [e_1/x][e_2/y]\varepsilon_3\end{array}}{\Gamma \vdash e_1.m(e_2) : \{\varepsilon\}\ [e_1/x][e_2/y]\tau_1}\ \text{(T-Method)}$$

$$\frac{\Gamma \vdash e : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\} \quad \mathsf{var}\ f : \tau \in \overline{\sigma}}{\Gamma \vdash e.f : \{\varepsilon\}\ [e/x]\tau}\ \text{(T-Field)} \qquad \frac{\Gamma \vdash e : \{\varepsilon_1\}\ \tau_1 \quad \Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash e : \{\varepsilon_2\}\ \tau_2}\ \text{(T-Sub)}$$

$$\frac{\Gamma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\} \quad \mathsf{var}\ f : \tau \in \overline{\sigma} \quad \Gamma \vdash e_2 : \{\varepsilon_2\}\ \tau \quad \varepsilon = \varepsilon_1 \cup \varepsilon_2}{\Gamma \vdash e_1.f = e_2 : \{\varepsilon\}\ [e_1/x]\tau}\ \text{(T-Assign)}$$

$\boxed{\Gamma \vdash d : \sigma}$

$$\frac{\begin{array}{c}\Gamma,\ x : \tau_1 \vdash e : \{\varepsilon_2\}\ \tau_2 \quad \Gamma,\ x : \tau_1 \vdash \varepsilon_1\ wf \\ \Gamma, x : \tau_1 \vdash \varepsilon_2 <: \varepsilon_1\end{array}}{\Gamma \vdash \mathsf{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 = e\ :\ \mathsf{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2}\ \text{(DT-Def)}$$

$$\frac{\Gamma \vdash x : \{\}\ \tau}{\Gamma \vdash \mathsf{var}\ f : \tau = x\ :\ \mathsf{var}\ f : \tau}\ \text{(DT-Var)} \qquad \frac{\Gamma \vdash \varepsilon\ wf}{\Gamma \vdash \mathsf{effect}\ g = \{\varepsilon\}\ :\ \mathsf{effect}\ g = \{\varepsilon\}}\ \text{(DT-Effect)}$$

Fig. 10. Wyvern static semantics.

are well formed. A field-declaration type is well formed if its type is well formed. Since an effect-declaration type has no right-hand side, it is trivially well formed. An effect-definition type is well formed if the effect set in its right-hand side is well formed. Finally, a bounded effect declaration is well formed if the upper bound or lower bound on the right-hand side is well formed. An effect set is well formed if, for every effect it contains, the definition of the effect doesn't form a cycle, the variable in the first part of the effect is well typed and the type of that variable contains either an effect-declaration or an effect-definition type, in which the effect name matches the effect name in the second part of the effect.

The $\Gamma \vdash safe(x.g, \varepsilon)$ judgment ensures that the definition of effect $x.g$ doesn't contain a cycle. The rules Safe-1, Safe-2, and Safe-3 are identical except the declaration of the effect type. The effect set $\varepsilon$ memorizes a set of effects that are defined by $x.g$. The rule ensures that those effects do not appear in the definition of $x.g$, therefore eliminating cycles in effect definition.

### 5.4 Static Semantics

Wyvern's static semantics is presented in Fig. 10. Expression type checking includes checking the effects that an expression may have, the set of which is denoted in a pair of curly braces between the colon and the type in the type annotation. Then, for expressions, the judgement reads that, in the variable typing context $\Gamma$, the expression $e$ is a well-typed expression with the effect set $\varepsilon$ and the type $\tau$.

A variable trivially has no effects. A new expression also has no effects because of the fact that fields may be initialized only using variables. A new object is well typed if all of its declarations are well typed.

A method call is well typed if the expression passed into the method as an argument is well typed, if the expression the method is called on is well typed, and if the expression's type contains a matching method-declaration type. In addition, bearing the appropriate variable substitutions,

the effect set annotating the method-declaration type must be well formed, and the effect set $\varepsilon$ in the method-call type must be a union of the effect sets of both expressions involved in the method call as well as the the effect set of the method-declaration type. The expressions that are being substituted are always the terminal runtime form, i.e., the expressions have been fully evaluated before they are substituted.

An object field read is well typed if the expression on which the field is dereferenced is well typed and the expression's type contains a matching field-declaration type. The effects of an object field type are those of the expression on which the field dereferencing is called.

A field assignment is well typed if the expression to which the field belongs is well typed and the expression's type has an appropriate field-declaration type, and if the expression in the right-hand side of the assignment is well typed. The effect set that a field assignment produces is a union between the effect sets the two expressions that are involved in the field assignment produce.

A type substitution of an expression may happen only if the expression is well typed using the original type, the original type is a subtype of the new type, and when the effect set of the original set is a subeffect of the effect of the new type. (Subeffecting is discussed in Section 5.5.1.)

None of the object declarations produce effects, and so object-declaration type-checking rules do not include an effect set preceding the type annotation. For declarations, the judgement reads that, in the variable typing context $\Gamma$, the declaration $d$ is a well-typed declaration with the type $\sigma$.

When type-checking a method declaration, the effect set annotating the method must be well formed in the overall typing context extended with the method argument. Furthermore, the effect annotating the method must be a supereffect of the effect the method body actually produced.

A field declaration is trivially well typed, and an effect declaration is well typed if the effect set that it is defined with is well formed in the given context.

## 5.5 Subtyping

$\boxed{\Gamma \vdash \varepsilon <: \varepsilon'}$

$$\frac{\varepsilon_1 \subseteq \varepsilon_2}{\Gamma \vdash \varepsilon_1 <: \varepsilon_2} \text{ (Subeffect-Subset)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g \leqslant \varepsilon \in \sigma \quad \Gamma \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2} \text{ (Subeffect-Upperbound)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g \geqslant \{\varepsilon\} \in \sigma \quad \Gamma \vdash \varepsilon_1 <: [n/y]\varepsilon \cup \varepsilon_2}{\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{n.g\}} \text{ (Subeffect-Lowerbound)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g = \{\varepsilon\} \in \sigma \quad \Gamma \vdash \varepsilon_1 <: [n/y]\varepsilon \cup \varepsilon_2}{\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{n.g\}} \text{ (Subeffect-Def-1)}$$

$$\frac{\Gamma \vdash n : \{y \Rightarrow \sigma\} \quad \texttt{effect}\ g = \{\varepsilon\} \in \sigma \quad \Gamma \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2} \text{ (Subeffect-Def-2)}$$

Fig. 11. Wyvern subeffecting rules.

*5.5.1 Subeffecting Rules.* As we already saw in the T-Sub, and DT-Def rules above and as we will see more in the upcoming Section 5.5.2, to compare two sets of effects, we use subeffecting rules, which are presented in Fig. 11. If an effect is a subset of another effect, then the former effect is a subeffect of the latter (Subeffect-Subset). If an effect set contains an effect variable that is declared with an upper bound, and the union of the rest of the effect set with the upper bound is a subeffect of another effect set, then the former effect set is a subeffect of the latter effect

$\boxed{size(\Gamma, \varepsilon) = n}$

$$\frac{}{size(\Gamma, \{\}) = 0} \text{ (SIZE-EMPTY)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g \in \sigma}{size(\Gamma, x.g) = 0} \text{ (SIZE-ABSTRACT)}$$

$$\frac{}{size(\Gamma, \overline{x.g}) = \Sigma_{x.g \in \overline{x.g}} size(\Gamma, x.g)} \text{ (SIZE-LIST)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g = \{\varepsilon\} \in \sigma}{size(\Gamma, x.g) = 1 + size(\Gamma, [x/y]\varepsilon)} \text{ (SIZE-DEF)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g \leqslant \{\varepsilon\} \in \sigma}{size(\Gamma, x.g) = 1 + size(\Gamma, [x/y]\varepsilon)} \text{ (SIZE-UPPERBOUND)}$$

$$\frac{\Gamma \vdash x : \{y \Rightarrow \sigma\} \quad \texttt{effect } g \geqslant \{\varepsilon\} \in \sigma}{size(\Gamma, x.g) = 1 + size(\Gamma, [x/y]\varepsilon)} \text{ (SIZE-LOWERBOUND)}$$

Fig. 12. Rules for determining the size of effect definitions.

set (SUBEFFECT-LOWERBOUND). If an effect set contains an effect variable that is declared with an lower bound, and the union of the rest of the effect set with the lower bound is a supereffect of another effect set, then the former effect set is a supereffect of the latter (SUBEFFECT-LOWERBOUND). If an effect set contains an effect variable that has a definition, and the union of the rest of the effect set with the definition of the variable is a supereffect of another effect set, then the former effect set is a supereffect of the latter (SUBEFFECT-DEF-1). Finally, if an effect set contains an effect variable that has a definition, and the union of the rest of the effect set with the definition of the variable is a subeffect of another effect set, then the former effect set is a subeffect of the latter (SUBEFFECT-DEF-2).

**Lemma 1.** *$size(\Gamma, \varepsilon)$ (Defined in Fig. 12) is finite.*

PROOF. By rules Safe-1, Safe-2, Safe-3, and Safe-4 in Fig. 9, the size of an arbitrary effect $x.g$ is bounded by the total number of effects in the context $\Gamma$. □

**Theorem 1.** *$\Gamma \vdash \varepsilon <: \varepsilon'$ is decidable.*

PROOF. The proof is by induction on $size(\Gamma, \varepsilon \cup \varepsilon')$.

BC Since size for both effect is 0, the only applicable rule for subeffecting is Subeffect-Subset. The rule only checks if $\varepsilon$ is a subset of $\varepsilon'$, therefore is decidable.

IS Assume the judgment $\Gamma \vdash \varepsilon <: \varepsilon'$ is derived from Subeffect-Upperbound. In the premise of this rule, we have $\Gamma \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2$. Since we extract the definition of $n.g$ to find $\varepsilon$, we have $size(\Gamma, [n/y]\varepsilon \cup \varepsilon_1 \cup \varepsilon_2) < size(\Gamma, \{n.g\} \cup \varepsilon_1 \cup \varepsilon_2)$. We can then use induction hypothesis to show the subeffecting judgment in the premise is decidable.

The inductive step for rules Subeffect-Lowerbound, Subeffect-Def-1, and Subeffect-Def-2 have the similar structure.

□

*5.5.2 Declarative Subtyping Rules.* Wyvern subtyping rules are shown in Fig. 13. Since, to compare types, we need to compare the effects in them using subeffecting, subtyping relationship is checked in a particular variable typing context. The first four object-subtyping rules and the S-REFL2 rule are standard. In S-DEPTH, since effects may contain a reference to the current object, to check the

$\boxed{\Gamma \vdash \tau <: \tau'}$

$$\frac{}{\Gamma \vdash \tau <: \tau} \text{ (S-Refl1)} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \text{ (S-Trans)}$$

$$\frac{\{x \Rightarrow \sigma_i^{i\in 1..n}\} \textit{ is a permutation of } \{x \Rightarrow \sigma'^{i\in 1..n}_i\}}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1..n}\} <: \{x \Rightarrow \sigma'^{i\in 1..n}_i\}} \text{ (S-Perm)}$$

$$\frac{}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1..n+k}\} <: \{x \Rightarrow \sigma_i^{i\in 1..n}\}} \text{ (S-Width)} \qquad \frac{\forall i,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i\in 1..n}\} \vdash \sigma_i <: \sigma'_i}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1..n}\} <: \{x \Rightarrow \sigma'^{i\in 1..n}_i\}} \text{ (S-Depth)}$$

$\boxed{\Gamma \vdash \sigma <: \sigma'}$

$$\frac{}{\Gamma \vdash \sigma <: \sigma} \text{ (S-Refl2)} \qquad \frac{\Gamma \vdash \tau'_1 <: \tau_1 \quad \Gamma \vdash \tau_2 <: \tau'_2 \quad \Gamma, x : \tau_1 \vdash \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash \mathsf{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 <: \mathsf{def}\ m(x : \tau'_1) : \{\varepsilon_2\}\ \tau'_2} \text{ (S-Def)}$$

$$\frac{}{\Gamma \vdash \mathsf{effect}\ g = \{\varepsilon\} <: \mathsf{effect}\ g} \text{ (S-Effect-1)} \qquad \frac{}{\Gamma \vdash \mathsf{effect}\ g \leqslant \varepsilon <: \mathsf{effect}\ g} \text{ (S-Effect-2)}$$

$$\frac{\Gamma \vdash \varepsilon <: \varepsilon'}{\Gamma \vdash \mathsf{effect}\ g = \{\varepsilon\} <: \mathsf{effect}\ g \leqslant \varepsilon'} \text{ (S-Effect-3)} \qquad \frac{\Gamma \vdash \varepsilon <: \varepsilon'}{\Gamma \vdash \mathsf{effect}\ g \leqslant \varepsilon <: \mathsf{effect}\ g \leqslant \varepsilon'} \text{ (S-Effect-4)}$$

$$\frac{}{\Gamma \vdash \mathsf{effect}\ g \geqslant \varepsilon <: \mathsf{effect}\ g} \text{ (S-Effect-5)} \qquad \frac{\Gamma \vdash \varepsilon' <: \varepsilon}{\Gamma \vdash \mathsf{effect}\ g = \{\varepsilon\} <: \mathsf{effect}\ g \geqslant \varepsilon'} \text{ (S-Effect-6)}$$

$$\frac{\Gamma \vdash \varepsilon' <: \varepsilon}{\Gamma \vdash \mathsf{effect}\ g \geqslant \varepsilon <: \mathsf{effect}\ g \geqslant \varepsilon'} \text{ (S-Effect-7)}$$

Fig. 13. Wyvern subtyping rules.

subtyping relationship between two type declarations, we extend the current typing context with the current object. Method-declaration typing is contravariant in the argument types and covariant in the return type. Furthermore, there must be a covariant-like relationship between the effect sets in the method annotations on the two method declarations: the effect set of the subtype method declaration must be a subeffect of the effect set of the supertype method declaration (S-Def). An effect definition or an effect declaration with bound is trivially a subtype of an effect declaration (S-Effect-1, S-Effect-2, S-Effect-5). An effect definition is a subtype of an effect declaration with upper bound if the definition is a subeffect of the upper bound (S-Effect-3). Similarly, an effect definition is a subtype of an effect declaration with lower bound if the definition is a supereffect of the lower bound (S-Effect-6). An effect declaration with upper bound is a subtype of the effect declaration with another upper bound if the former upper bound is a subeffect of the latter upper bound (S-Effect-4). Finally, an effect declaration with lower bound is a subtype of the effect declaration with another lower bound if the former upper bound is a supereffect of the latter upper bound (S-Effect-7).

$\boxed{\Gamma \vdash \tau <: \tau'}$

$$\frac{\exists\ \textit{an injection}\ p : \{1...n\} \mapsto \{1...m\}, \quad \forall i \in 1...n,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma_{p(i)} <: \sigma'_i}{\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1...m}\} <: \Gamma \vdash \{x \Rightarrow \sigma'^{i\in 1...n}_i\}} \text{ (S-Alg)}$$

Fig. 14. Algorithmic Subtyping

*5.5.3 Algorithmic Subtyping Rules.* The S-Alg rule encodes the S-Refl-1, S-Perm, S-Depth, and S-Width rule using an injective function $p$. The subtyping rules of declaration types are identical to the declarative subtyping. We prove that S-Trans rules is emissible in theorem 2. Since subtyping rules object types and declaration types are syntax-directed, the subtyping of our effect system is decidable.

**Theorem 2.** *(Transitivity of algorithmic subtyping)*
*If* $\Gamma \vdash \tau_1 <: \tau_2$ *and* $\Gamma \vdash \tau_2 <: \tau_3$, *then* $\Gamma \vdash \tau_1 <: \tau_3$.
*If* $\Gamma \vdash \sigma_1 <: \sigma_2$ *and* $\Gamma \vdash \sigma_2 <: \sigma_3$, *then* $\Gamma \vdash \sigma_1 <: \sigma_3$.

## 5.6 Dynamic Semantics and Type Soundness

| | | | |
|---|---|---|---|
| $n$ | ::= | $x \mid l$ | *names* |
| $e$ | ::= | $n$ | *expressions* |
| | \| | $\texttt{new}(x \Rightarrow \overline{d})$ | |
| | \| | $e.m(e)$ | |
| | \| | $e.f$ | |
| | \| | $e.f = e$ | |
| $\varepsilon$ | ::= | $\overline{n.g}$ | *effects* |
| $d$ | ::= | $\texttt{def}\ m(x : \tau) : \{\varepsilon\}\ \tau = e$ | *declarations* |
| | \| | $\texttt{var}\ f : \tau = n$ | |
| | \| | $\texttt{effect}\ g = \{\varepsilon\}$ | |
| $\tau$ | ::= | $\{x \Rightarrow \overline{\sigma}\}$ | *object type* |

| | | | |
|---|---|---|---|
| $\sigma$ | ::= | $\texttt{def}\ m(x : \tau) : \{\varepsilon\}\ \tau$ | *declaration types* |
| | \| | $\texttt{var}\ f : \tau$ | |
| | \| | $\texttt{effect}\ g$ | |
| | \| | $\texttt{effect}\ g \geqslant \{\varepsilon\}$ | |
| | \| | $\texttt{effect}\ g \leqslant \{\varepsilon\}$ | |
| | \| | $\texttt{effect}\ g = \{\varepsilon\}$ | |
| $\Gamma$ | ::= | $\varnothing \mid \Gamma,\ x : \tau$ | *var. typing context* |
| $\mu$ | ::= | $\varnothing \mid \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}$ | *store* |
| $\Sigma$ | ::= | $\varnothing \mid \Sigma,\ l : \tau$ | *store typing context* |
| $E$ | ::= | $[\,]$ | *evaluation context* |
| | \| | $E.m(e)$ | |
| | \| | $l.m(E)$ | |
| | \| | $E.f$ | |
| | \| | $E.f = e$ | |
| | \| | $l.f = E$ | |

Fig. 15. Wyvern's object-oriented core syntax with dynamic forms.

*5.6.1 Object-Oriented Core Syntax.* Fig. 15 shows the version of the syntax of Wyvern's object-oriented core that includes dynamic semantics. Specifically, expressions include locations $l$, which variables in effects resolve to at run time. We also use a store $\mu$ and its typing context $\Sigma$. Finally, to make the dynamics more compact we use an evaluation context $E$.

*5.6.2 Changes in Static Semantics.* Type checking a location (T-Loc) and a field declaration (DT-Var) is straightforward, and we also need to ensure that the store is well-formed and contains objects that respect their types.

*5.6.3 Dynamic Semantics.* The dynamic semantics that we use for Wyvern's effect system is shown in Fig. 17 and is similar to the one described in prior work [Melicher et al. 2017]. In comparison to the prior work, this version of Wyvern's dynamic semantics has fewer rules, and the E-Method rule is simplified.

The judgement reads the same as before: given the store $\mu$, the expression $e$ evaluates to the expression $e'$ and the store becomes $\mu'$. The E-Congruence rule still handles all non-terminal forms. To create a new object (E-New), we select a fresh location in the store and assign the object's definition to it. Provided that there is an appropriate method definition in the object on which a method is called, the method call is reduced to the method's body (E-Method). In the method's body, the locations representing the method argument and the object on which the method is called are substituted for corresponding variables. An object field is reduced to the value held in it (E-Field), and when an object field's value changes (E-Assign), appropriate substitutions are made in the object's declaration set and the store.

$\boxed{\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau}$

$$\ldots \quad \frac{l : \tau \in \Sigma}{\Gamma \mid \Sigma \vdash l : \{\}\ \tau}\ \text{(T-Loc)}$$

$\boxed{\Gamma \mid \Sigma \vdash d : \sigma}$

$$\ldots \quad \frac{\Gamma \mid \Sigma \vdash n : \{\}\ \tau}{\Gamma \mid \Sigma \vdash \mathsf{var}\ f : \tau = n\ :\ \mathsf{var}\ f : \tau}\ \text{(DT-Var)}$$

$\boxed{\mu : \Sigma}$

$$\frac{\forall l \mapsto \{x \Rightarrow \overline{d}\} \in \mu,\ \forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ x : \{x \Rightarrow \overline{\sigma}\} \mid \Sigma \vdash d_i : \sigma_i}{\mu : \Sigma}\ \text{(T-Store)}$$

Fig. 16. Wyvern static semantics affected by dynamic semantics.

$\boxed{\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle}$

$$\frac{\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle}{\langle E[e] \mid \mu\rangle \longrightarrow \langle E[e'] \mid \mu'\rangle}\ \text{(E-Congruence)} \qquad \frac{l \notin dom(\mu)}{\langle \mathsf{new}(x \Rightarrow \overline{d}) \mid \mu\rangle \longrightarrow \langle l \mid \mu, l \mapsto \{x \Rightarrow \overline{d}\}\rangle}\ \text{(E-New)}$$

$$\frac{l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu \quad \mathsf{def}\ m(y : \tau_1) : \{\varepsilon\}\ \tau_2 = e \in \overline{d}}{\langle l_1.m(l_2) \mid \mu\rangle \longrightarrow \langle [l_2/y][l_1/x]e \mid \mu\rangle}\ \text{(E-Method)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\} \in \mu \quad \mathsf{var}\ f : \tau = l_1 \in \overline{d}}{\langle l.f \mid \mu\rangle \longrightarrow \langle l_1 \mid \mu\rangle}\ \text{(E-Field)}$$

$$\frac{l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu\ \mathsf{var}\ f : \tau = l \in \overline{d} \qquad \overline{d'} = [\mathsf{var}\ f : \tau = l_2/\mathsf{var}\ f : \tau = l]\overline{d}\ \mu' = [l_1 \mapsto \{x \Rightarrow \overline{d'}\}/l_1 \mapsto \{x \Rightarrow \overline{d}\}]\mu}{\langle l_1.f = l_2 \mid \mu\rangle \longrightarrow \langle l_2 \mid \mu'\rangle}\ \text{(E-Assign)}$$

Fig. 17. Wyvern dynamic semantics.

*5.6.4 Type Soundness.* We prove the soundness of the effect system presented above using the standard combination of progress and preservation theorems. Proof to these theorems can be found in Appendix B.

**Theorem 3** (Preservation). *If* $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, $\mu : \Sigma$, *and* $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$, *then* $\exists \Sigma' \supseteq \Sigma$, $\mu' : \Sigma'$, $\exists \varepsilon'$, *such that* $\Gamma \vdash \varepsilon' <: \varepsilon$, *and* $\Gamma \mid \Sigma' \vdash e' : \{\varepsilon'\}\ \tau$.

**Theorem 4** (Progress). *If* $\varnothing \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$ *(i.e., e is a closed, well-typed expression), then either*

*(1) e is a value (i.e., a location) or*

*(2)* $\forall \mu$ *such that* $\mu : \Sigma$, $\exists e', \mu'$ *such that* $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$.

## 5.7 Authority

Our definition of authority is based on prior research [Drossopoulou et al. 2016; Miller 2006] and says that authority is the ability to operate on resources. Using the extra information that effect members and annotations provide, we can now talk about authority of modules (and objects) in an application.

*5.7.1 Authority Safety.* We define an authority-safe programming language as one that provides a way for a software developer to specify and limit modules' (or objects') authority over other modules (or objects) using a set of well-defined rules. Through examples in Sections 2–4, we

$\boxed{auth(\texttt{new}(x \Rightarrow \overline{d}))}$

$$auth(\texttt{new}(x \Rightarrow \overline{d})) = \bigcup_{d \in \overline{d}} auth(d) \text{ (Auth-Object)}$$

$\boxed{auth(d)}$

$$auth(\texttt{def } m(x : \tau_1) : \{\varepsilon\}\ \tau_2 = e) = \varepsilon \cup auth(\tau_2) \text{ (Auth-Def)}$$
$$auth(\texttt{var } f : \tau = n) = auth(\tau) \text{ (Auth-Var)}$$
$$auth(\texttt{effect } g = \{\varepsilon\}) = \varnothing \text{ (Auth-Effect)}$$

$\boxed{auth(\tau)}$

$$\frac{\tau = \{x \Rightarrow \overline{\sigma}\}}{auth(\tau) = \bigcup_{\sigma \in \overline{\sigma}} auth(\sigma)} \text{ (Auth-Type)}$$

$\boxed{auth(\sigma)}$

$$auth(\texttt{def } m(x : \tau_1) : \{\varepsilon\}\ \tau_2) = \varepsilon \cup auth(\tau_2) \text{ (Auth-DefType)}$$
$$auth(\texttt{var } f : \tau) = auth(\tau) \text{ (Auth-VarType)}$$
$$auth(\texttt{effect } g) = \varnothing \text{ (Auth-AbsEffectType)}$$
$$auth(\texttt{effect } g = \{\varepsilon\}) = \varnothing \text{ (Auth-ConEffectType)}$$
$$auth(\texttt{effect } g \geqslant \{\varepsilon\}) = \varnothing \text{ (Auth-UBEffectType)}$$
$$auth(\texttt{effect } g \leqslant \{\varepsilon\}) = \varnothing \text{ (Auth-LBEffectType)}$$

Fig. 18. Rules defining authority of an object.

illustrated how a software developer could use effect annotations to specify and control modules' authority. Our formal system ensures that the program behavior adheres to the rules specified by the software developer. Specifically, Wyvern's static semantics (Section 5.4) checks that effect annotations correspond to the effects produced by each method body, and the preservation theorem 3 guarantees that effects produced during execution adhere to the effect annotations in the program, because preservation states that the effect of a program only decreases as a program executes, and never increases (the effect may decrease because it is conditional and the guarding condition is not fulfilled, or because the effect takes place and the remainder of the program does not have that effect). Then, since we proved the type soundness of Wyvern's effect system, we proved Wyvern authority safe.

*5.7.2 Authority of an Object.* A basic notion in the authority analysis of an application is the notion of an object's authority, which we define next.

**Definition 1** (Authority of an object)**.** The authority of an object is a set of effects that the object's methods and fields can produce.

This definition is "outward facing" in a sense that it helps reasoning about authority of objects that use the current object. We chose such definition because it seems to be more useful in a security analysis. For example, if an application's programming interface allows plugins to access a specific module (e.g., the `logger` module described in Fig. 2), it is useful to be able to determine what effects a plugin could produce by using that module, accessing its fields and calling methods on it.

Formally, we represent an object's authority as a set of *auth* rules, shown in Fig. 18. An object's authority (Auth-Object) is the authority of the object's declarations. Authority of a method declaration (Auth-Def) is the effects that the method produces during execution and also the authority of objects of the method's return type. The reason for including the latter authority component is that, whenever the method is called, an object of the return type is returned to and may be operated on by the caller, thus increasing the caller's authority. For the same reason, authority of an object's field (Auth-Var) is the authority of objects of the field's type. An effect declaration carries no authority (Auth-Effect).

Authority of objects of a particular type (Auth-Type) is authority of the type's declarations. Authority of a method-declaration type (Auth-DefType), a field-declaration type (Auth-VarType), and a concrete-effect-declaration type (Auth-ConEffectType) is similar to the authority of corresponding declarations in an object. An abstract-effect-declaration type produces no authority (Auth-AbsEffectType). Moreover, declaration types for bounded effects produce no authority (Auth-UBEffectType, Auth-LBEffectType).

As an example of how these rules can be applied in practice, if we look at the `logger` module presented in Fig. 2, using the *auth* rules, we can determine that `logger` has authority to read the log file (the `ReadLog` effect) and to update the log file (the `UpdateLog` effect), but no other authority. Although this example may seem trivial, knowing an object's authority is useful if we analyze objects that are more complex than the `logger` object and also for object comparison, like we demonstrate in the next section.

*5.7.3 Authority Attenuation.* Introduced in Mark Miller's dissertation [Miller 2006], the notion of authority attenuation can be described as follows. If a module (or an object) accesses a resource and produces less than the total possible set of operations on that resource, we say that that module (or object) *attenuates* the resource. For example, consider the modules in Figs. 2, 3, and 5. We observe that, while the `file` module can have a number of effects (`Read`, `Write`, `Append`, etc.), the `logger` module produces only two of `file`'s effects (`Read` and `Append`). Then, any module that uses `logger` and does not have access to the `file` module (e.g., the `codeCompletion` plugin module) can produce on `file` *at most* the two effects `logger` can produce. Thus, the `logger` module attenuates the `file` module by giving access to only a subset of `file`'s effects.

To aid a security analyst in a formal security analysis of an application, we formalized the notion of authority attenuation. Importantly, our definition of authority attenuation is static. We only examine an object's code and do not know which specific objects the object uses at run time. Instead, we can talk about objects of a specific *type* that the object uses. Our definition of authority attenuation benefits from this since we can talk about *groups of objects* any object in which is attenuated. For example, using our static definition of authority attenuation, instead of knowing that the `logger` module attenuates the `file` module (which is of type `File`), we know that `logger` attenuates *all* objects of type `File`.

In essence, our formal definition says that if we let $F_1$ be the set of effects that represents an object's authority and $F_2$ be the set of effects that represents authority of objects of a specific type. Then, if $F_1$ and $F_2$ share at least one effect and there is at least one effect that is in $F_2$ but not in $F_1$, we say that the object attenuates objects of that type. For example, if we let $F_1$ be the set of effects that represents the `logger`'s authority and $F_2$ be the set of effects that represents authority of objects of type `File`. Then, if $F_1$ and $F_2$ share at least one effect and there is at least one effect that is in $F_2$ but not in $F_1$, we say that `logger` attenuates objects of type `File`. Formally, we write these conditions as follows.

**Definition 2** (Authority Attenuation[4])**.** An object $o$ attenuates objects of type $\tau$, if $F_1 = tLookup(\Gamma, \tau, auth(o))$, $F_2 = tLookup(\Gamma, \tau, auth(\tau))$, $F_1 \cap F_2 \neq \emptyset$, and $F_2 \setminus F_1 \neq \emptyset$.

First, using the *auth* rules, shown in Fig. 18, we find authority of object $o$ and of objects of type $\tau$. Then, we use the *tLookup* rules, shown in Fig. 19, to "normalize" the two effect sets, thus making it possible to compare them. Finally, we compare the two effect sets.

[4]It is possible to create a more general formal definition of authority attenuation by, instead of considering one object that attenuates objects of a specific type, considering objects of one type that attenuate objects of another type. This version of the authority attenuation definition is presented in Appendix **??**.

$\boxed{tLookup(\Gamma, \tau, \overline{x.g})}$

$$tLookup(\Gamma, \tau, \overline{x.g}) = \bigcup_{x.g \in \overline{x.g}} tLookup(\Gamma, \tau, x.g) \text{ (tLookup)}$$

$\boxed{tLookup(\Gamma, \tau, x.g)}$

$$\frac{\Gamma \vdash x : \{\}\ \tau}{tLookup(\Gamma, \tau, x.g) = \tau.g} \text{ (tLookup-Stop-1)}$$

$$\frac{\Gamma \vdash x : \{\}\ \tau' \quad \tau' \neq \tau \quad \tau' = \{y \Rightarrow \overline{\sigma}\} \quad (\texttt{effect}\ g \in \overline{\sigma} \vee \texttt{effect}\ g \geqslant \{\varepsilon\} \in \overline{\sigma})}{tLookup(\Gamma, \tau, x.g) = \tau'.g} \text{ (tLookup-Stop2)}$$

$$\frac{\Gamma \vdash x : \{\}\ \{y \Rightarrow \overline{\sigma}\} \quad \tau \neq \{y \Rightarrow \overline{\sigma}\} \quad (\texttt{effect}\ g = \{\varepsilon\} \in \overline{\sigma} \vee \texttt{effect}\ g \leqslant \{\varepsilon\} \in \overline{\sigma})}{tLookup(\Gamma, \tau, x.g) = tLookup(\Gamma, \tau, [x/y]\varepsilon)} \text{ (tLookup-Recurse)}$$

Fig. 19. Wyvern effect-lookup rules that target a specific type.

The *tLookup* rules support the static nature of our definition of authority attenuation. They resolve effects to lower-level effects by "searching" for effects of an object of a particular *type* and stopping when an object of that type is found. When we apply *tLookup* to a set of effects, we apply *tLookup* to each effect in that set (tLookup). If the type that we are looking for is the type of the current object (tLookup-Stop), we return the "normalized" form of the effect, which differs from the original form in that we substitute the variable name with the type name. If we encounter an abstract effect (tLookup-Stop2), we return the "normalized" form of that effect that uses the type of the current object. Otherwise, the effect is concrete, and we proceed by examining the effect's definition (tLookup-Recurse).

As an example, let us apply our definition of authority attenuation to the `logger` module and the objects of type `File` (e.g., the `file` module) from Figs. 2 and 3 respectively. Using the *auth* and *tLookup* rules on the `logger` object, we find that `logger`'s authority is $F_1 = \{File.Read, File.Append\}$. Similarly, we find that the authority of objects of type `File` is $F_2 = \{File.Read, File.Write, File.Append, \ldots\}$. Then, comparing the two sets, we have $F_1 \cap F_2 = \{File.Read, File.Append\} \neq \emptyset$ and $F_2 \setminus F_1 = \{File.Write, \ldots\} \neq \emptyset$. Thus, by our definition, the `logger` module attenuates modules of type `File`.

**Definition 3** (Authority Attenuation (more generally))**.** Objects of type $\tau_1$ attenuate objects of type $\tau_2$, if

(1) $F_1 = tLookup(\Gamma, \tau, auth(\tau_1))$, $F_2 = tLookup(\Gamma, \tau, auth(\tau_2))$,
(2) $F_1 \cap F_2 \neq \emptyset$, and
(3) $F_2 \setminus F_1 \neq \emptyset$.

This definition essentially says that if we let $F_1$ be the set of effects that represents authority of objects of one type and $F_2$ be the set of effects that represents authority of objects of another type. Then, if $F_1$ and $F_2$ share at least one effect and there is at least one effect that is in $F_2$ but not in $F_1$, we say that objects of the former type attenuate objects of the latter type.

## 6 CASE STUDY: AN EXTENSIBLE TEXT-EDITOR APPLICATION

Effect checking was implemented as part of Wyvern.[5] To evaluate our effect-system design, we used Wyvern to create an extensible text-editor application and plugins for it, similar to the running example described earlier.[6]

[5] https://github.com/wyvernlang/wyvern
[6] https://anonymous.4open.science/repository/c7727f9f-6309-4c1e-b382-5b93e20cc451/examples/text-editor/

### 6.1 Application Description

The text-editor application provides basic text-editing functionality. When started, the text-editor window has a text area where the user may enter or edit text. The title bar shows the path to the currently opened document or "Untitled" if the document have not been saved yet. The menu bar has three options and allows users to perform operations on files, perform editing operations on the text in the text area, or run plugins. The main application module is called `textEditor` and is of type `TextEditor`.

```
1 resource type Plugin
2   effect Run
3   def getName(): {} String
4   def run(): {this.Run} Unit
```

Fig. 20. The `Plugin` type that each text editor's plugin must implement.

We implemented the following three plugins:

- `darkTheme` sets the theme of the text editor to have a dark background and light text,
- `questionnaireCreator` extracts questions from the currently opened document and creates a questionnaire in a separate file, and
- `wordCount` counts the number of words in the currently opened document and displays that number to the user in a pop-up window.

All plugins must implement the `Plugin` type, shown in Fig. 20.

### 6.2 Observations and Discussion

During the implementation of the text-editor application, we made several observations that stem from the way we designed Wyvern's effect systems, which we present and discuss next.

*6.2.1 Effect Aggregation.* Table 1 shows the average number (geometric mean[7]) of effects in each effect set used in the implementation. This aspect speaks to the amount of boilerplate code that the effect-aggregation feature of our effect-system design eliminates.

The average number of effects in the effect-definition sets is much lower for the text editor than for the plugins, which signals that effects declared in the text editor are usually composed of fewer effects than those declared in plugins. There are at least two reasons for that. The main reason is that text editor's methods frequently use only one resource each and perform only one operation on it, whereas, in a plugin, the `run` method tends to use all the resources that the plugin has access to. Another, minor reason is that the `textEditor` module defines an effect, called `SaveFile`, whose definition consists of four effects, which is then used as a shorthand in defining two out of seven `textEditor`'s effects.

In contrast to effect definitions, the difference between the average numbers of effects in effect-annotation sets in the text editor and the plugins is insignificant, and the numbers are low. For the text-editor application, the reason is that the same `SaveFile` is used to annotate five out of fifteen (i.e., one third of) `textEditor`'s methods. In addition, three more `textEditor`'s methods have empty effect annotations. For the plugins, the reason is that there is only one method (the `run` method) that has an effect annotation with an effect (`Plugin`'s `Run` effect) in it, and the rest of the methods have empty effect annotations.

| | Definitions | Annotations and parameters |
|---|---|---|
| Text editor | 1.2 | 1.1 |
| Plugins | 3.6 | 0.8 |
| Overall | 1.3 | 1.0 |

Table 1. The average number (geometric mean) of effects per an effect set.

Overall, these observations imply that the effect-aggregation feature has its merit and indeed serves to reduce the amount of effect-related code.

[7] To handle zeros in the data, we added one to each value, calculated the mean, and then subtracted one from the result.

| | LoC | Effect declarations | | Effect annotations | | Effect parameters | | Total | |
|---|---|---|---|---|---|---|---|---|---|
| Text editor | 250 | 38 | (15%) | 56 | (22%) | 3 | (1%) | 97 | (39%) |
| Plugins | 110 | 3 | (3%) | 12 | (11%) | 6* | (5%) | 21 | (19%) |
| Total | 360 | 41 | (11%) | 68 | (19%) | 9 | (3%) | 118 | (33%) |

Table 2. The effect-annotation overhead in the text-editor application and its plugins. For the plugins, the number of lines that contain effect parameters, marked with an asterisk, includes the lines where plugins are instantiated that are located in the text editor's code.

*6.2.2 Effect-Annotation Overhead.* Table 2 presents a higher-level picture about the effect-annotation overhead. Overall, the effect-annotation overhead comes from three sources: effect declarations, effect annotations on methods, and effect parameters. The important distinction among these types of annotation overhead is that effect declarations require adding new lines of code to the implementation, whereas effect annotations and effect parameters change the lines of code that would still exist in unannotated code.

Incorporating effects into the text editor's code base led to an 11% increase in its size and affected 22% of (the enlarged version of) it, and so, overall, 33% of code was affected by the inclusion of the effect information. The overhead is lower for the plugins than for the text-editor application itself. The reason for this difference is that the text-editor application accesses and operates on more resources than any one plugin does, and also the application defines the effects that the plugins may have. In contrast, all three plugins define and use only one effect, `Run`, that involves using resources (and one more empty effect), and also, none of the plugins introduce new effects that would be used only by the plugin itself. Based on this pattern, we do not expect effect annotations to be a deterrent to implementing plugins, and we expect the ratio of affected lines to go down as more plugins are added.

We did see one source of verbosity from effects. Since the `TextEditor` type is agnostic to its plugins, it has an abstract effect member `Run` that represents the effects that its plugins have. When the text editor runs a plugin, it incurs the `Run` effect. This saves annotation from within the `TextEditor`, but when we instantiate a `TextEditor` to be run with a specific set of plugins, we must parameterize it in a way that binds the `Run` effect to the set of all the effects of all of its plugins. In our example, this is 7 separate effects, and there might be even more if there were additional plugins. This verbose parameterized type also appears in the module header of the `textEditor` implementation. While this creates a couple of very long lines of code (e.g. 192 characters in the type instantiation in `main`), we view it as a good tradeoff, given that this large set of effects is encapsulated by the abstract `Run` effect everywhere else.

*6.2.3 Information Hiding and Polymorphism.* There are three examples that we observed in the text-editor application. The first example is in the plugin modules. Different plugins naturally have different effects; for example, the `darkTheme` and `questionnaireCreator` plugins both use the logging module, but otherwise have disjoint effects. We defined an abstract effect `Run` in the `Plugin` module, which is then given a different concrete definition in each concrete plugin implementation.

The second example is in the logging module. Currently, the `logger` module is implemented using the file system and stores the log file locally in a file. In the future, the text editor can be made distributed, and the logging module could maintain a log file which is stored somewhere else on the network (e.g., as was suggested in Section 4.1). Due to effect abstraction, this change is easily accommodated in the current version of the text editor's code. As long as the new, distributed logger implements the `Logger` type, the modules that use `logger` are not affected by the substitution.

The third example uses the effect-hierarchy feature of our effect system. Similarly to the `RemoteLogger` example in Section 4.1, we use an effect hierarchy to hide the definitions of the UI-related effects. Namely, we define the `UIEffects` type, which specifies a lower bound for each UI-related effect, and then ascribe this type to the `uiEffects` pure module that defines those effects.

In addition, our design can accommodate one more possible change. Currently, the `logger` module only appends to the log file, thus producing the `Append` effect on the `logFile` module, which is reflected in the definition of `logger`'s `Update` effect. Alternatively, `logger` could write to the log file aggregating the information that has been already logged, e.g., substituting "X action occurred. X action occurred." with "X action occurred 2 times." In such a case, `logger` would produce the `Write` effect on the `logFile` module, and the definition of `logger`'s `Update` effect would change accordingly. Due to effect abstraction, there would be no difference for the modules that use the `logger` module, which would still produce `logger`'s `Update` effect.

*6.2.4 Controlling Operations Performed on Modules.* In the text-editor application, plugins may be written by some third party, and thus their code is untrusted. To verify that plugins make no illegal calls, we need to check the effect annotations on the plugins and analyze the legitimacy of the effects produced on each text-editor's module that plugins access. For example, the `questionnaireCreator` plugin produced the `Update` effect on `logger`, the `Append` and `Write` effects on `fileSystem`, and the `Read` effect on `textArea`. These effects are congruent with `questionnaireCreator`'s expected functionality: the plugin produces the `Update` effect on `logger` to update the log file, the `Write` and `Append` effects on `fileSystem` to create a new file containing the resulting questionnaire and to append to it when a question is encountered in the original text, and the `Read` effect on `textArea` to read in the current version of the opened document. If `questionnaireCreator` had any more effects, those effects would have been unauthorized. Thus, all the effects that the plugin produces are legitimate. We performed a similar verification on the other plugins and determined that they are given access to the minimal number of text editor's modules and, on those, they perform only the necessary operations.

In addition, our effect system allows expressing the intent that a method may not produce any effects, which is then enforced by Wyvern's effect system. During the implementation of the text editor, we used this feature when defining the `Plugin` type (Fig. 20). We added a method, called `getName`, that returns the plugin's name, so that the plugin can be added to the text editor's user menu. All that `getName` needs to do is to return a `String` with the plugin's name, and thus the method must produce no effects. To enforce this restriction, we annotated the method with `{}`, i.e., an empty effect set, which achieved the desired behavior.

*6.2.5 Designating Important Resources Using Globally Available Effects.* While implementing the text-editor application, to define the effects of the UI-related modules, we made a design choice to use globally available effects (discussed in detail in Section 4.3). We defined the UI-related globally available effects in the pure module called `uiEffects` and used them throughout the code, hence designating the importance of tracking effects on the UI and making it more obvious where those effects are produced in the code.

*6.2.6 Authority Attenuation.* Section 5.7.3 describes how our effect system allows formalizing authority attenuation. In practice, as suggested in Section 4.4, since method effect annotations expose the information about how resources are used, a software developer is able to identify occurrences of authority attenuation by looking at modules' interfaces.

In the text-editor application, by examining only module interfaces, we were able to determine that the `logger` module attenuates the `logFile` object. While `logFile` has three effects: `Read`, `Write`, and `Append`, `logger` produces only the `Append` effect. This means that the `logger` module allows for only

a limited set of effects to be produced on `logFile`, thus attenuating it. Considering the structured nature of module interfaces, we believe that it is feasible to automate this discovery process.

*6.2.7 Effect Hierarchy.* In the text editor, we used the effect-hierarchy feature of our effect system twice. The first use case is in hiding the definitions of the UI-related effects which is discussed in Section 6.2.3. The second use case is in refining the `Plugin` type (Fig. 20) specifically for the text editor's theme plugins. We created a new type, called `ThemePlugin`, which is identical to the original `Plugin` type, except for its `Run` effect being lower-bounded by the UI effects necessary for updating the way the UI looks. We then used the `ThemePlugin` type when adding theme-related plugins to the text editor.

*6.2.8 Controlling FFI Effects.* The implementation of the text editor requires calling into several Java methods through an FFI. Our effect system expects all Wyvern methods calling into the Java FFI to be annotated with the `system.FFI` effect. Therefore, whenever a Wyvern method calling into the Java FFI was not annotated with a proper effect, the compiler rejected the program. Moreover, to have a fine-grained control of different kinds of FFI effects, based on `system.FFI`, we defined higher-level UI effects, such as `ShowDialog` and `ReadTextArea`, in the `uiEffects` module. This way we can reason about the effects of Wyvern methods that call into different Java methods separately.

*6.2.9 Additional Validation of the Wyvern Effect System .* [Fish et al. 2020] describes the application of our effect system to the Wyvern standard library. The paper presents a case study of a small standard I/O library seeking to use the effect system of Wyvern for tracking the secure use of resources. The study suggests that the effect system of Wyvern is indeed practicable and useful, and thus potentially promising for inclusion in other future language designs.

## 7 RELATED WORK

**Origins of Effect Systems.** Effect Systems were originally proposed by Lucassen [1987] to track reads and writes to memory, and then Lucassen and Gifford [1988] extended this effect system to support polymorphism. Effects have since been used for a wide variety of purposes, including exceptions in Java [Kiniry 2006] and asynchronous event handling [Bračevac et al. 2018]. Turbak and Gifford [2008] previously proposed effects as a mechanism for reasoning about security, which is the main application that we discuss.

**Denotational vs. Descriptive Effects.** Filinski [2010] makes a distinction between two strands of work on effects. A *denotational* approach, which includes algebraic effects, defines the semantics of computational effects based on primitives. A *restrictive* approach (e.g., Java's checked exceptions) takes effects that are already built into the language–such as reading and writing state or exceptions–and provides a way to restrict them. In this terminology, our approach is restrictive, rather than denotational.

**Prior Work on Bounded Effect Polymorphism.** A limited form of bounded effect polymorphism were explored by Trifonov and Shao [1999], who bound effect parameters by the resources they may act on; however, the bound cannot be another aribrary effect, as in our system. Long et al. [2015] use a form of bounded effect polymorphism internally but do not expose it to users of their system.

**Defining Application-Specific Effects.** Marino and Millstein [2009] discuss an effect system in which application-specific effects can be defined. One of their examples is system calls that can block, but their design does not provide the benefit of a semantic tie-in to the foreign function interface, as ours does.

**Path-Dependent Effects** JML's data groups [Leino et al. 2002] have some superficial similarities to Wyvern's effect members. Data groups are identifiers bound in a type that refer to a collection

of fields and other data groups. They allow a form of abstract reasoning, in that clients can reason about reads and writes to the relevant state without knowing the underlying definitions. Data groups are designed specifically to capture the modification of state, and it is not obvious how to generalize them to other forms of effects.

The closest prior work on path-dependent effects, by Greenhouse and Boyland [1999], allows programmers to declare regions as members of types; this supports a form of path-dependency in read and write effects on regions. Our formalism expresses path-dependent effects based on the type theory of DOT [Amin et al. 2014], which we find to be cleaner and easier to extend with the unique bounded abstraction features of our system. Amin et al.'s type members can be left abstract or refined by upper or lower bounds, and were a direct inspiration for our work on bounded abstract effects.

**Subeffecting.** Some effect systems, such as Koka [Leijen 2014], provide a built-in set of effects with fixed sub-effecting relationships between them. Rytz et al. [2012] supports more flexibility via an extensible framework for effects. Users can plug in their own domain of effects, specifying an effect lattice representing sueffecting relationships. Each plugin is monolithic. In contrast, our effect members allow new effects to be incrementally added and related to existing effects using declared subeffect bounds.

**Algebraic Effects, Generativity, and Abstraction.** Algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009] are a way of implementing certain kinds of side effects such as exceptions and mutable state in an otherwise purely functional setting. As described above, algebraic effects fall into the "denotational" rather than "descriptive" family of effects work; these lines of work are quite divergent, and it is often unclear how to translate technical ideas from one setting to the other. However, certain papers explore parallels to our work, despite the major contextual differences.

Bračevac et al. [Bračevac et al. 2018] use algebraic effects to support asynchronous, event-based reactive programs. They need to use a different algebraic effect for each join operation that correlates events; thus, they want effects to be generative. This generativity is at a per-module level, however, whereas our work supports per-object generativity.

[Zhang and Myers 2019] describe a design for algebraic effects that preserves abstraction in the sense of parametric functions: if a function does not statically know about an algebraic effect, that effect tunnels through that function. This is different from our form of abstraction, in which the definition of an effect is hidden from clients.

[Biernacki et al. 2019] discuss how to abstract algebraic effects using existentials. The setting of algebraic effects makes their work quite different from ours: their abstraction hides the "handler" of an effect, which is a dynamic mechanism that actually implements effects such as exceptions or mutable state. In contrast, our work allows a high-level effect to be defined in terms of zero or more lower-level effects, and our abstraction mechanism allows the programmer to hide the lower-level effects that constitute the higher-level effect. Our system, unlike algebraic effect systems, is purely static. We do not attempt to implement effects, but rather give the programmer a system for reasoning about side effects on system resources and program objects. It is not clear that defining a high-level effect that encapsulates multiple low-level events is sensible in the setting of algebraic effects, since this would require merging effect implementations that could be as diverse as mutable state and exception handling. It is also not clear how Biernacki et al.'s abstraction of algebraic effects could apply to the security scenarios we examine in Section 4, since some of our scenarios rely critically on abstracting lower-level events as higher-level ones.

**Comparison to the Effekt Library** The Effekt library [BRACHTHÄUSER et al. 2020] explores algebraic effects as a library of the Scala programming language. Since their effect system is built on type members of Scala, it bears some similarities to our system (e.g. their use of type members

and our use of effect members). Despite superficial similarities, their work is largely orthogonal to our contributions due to the following comparisons:

(1) Our goal is to check the effects of general purpose code. In contrast, Brachthäuser's approach requires all effect-checked code to be written in a monad. This is required to support control effects (i.e. prescriptive effects), but it is not an incidental difference: it is also an integral part of their static effect checking system, because monads are the way that they couple Scala's type members (which provide abstraction and polymorphism) to effects. Their paper therefore, does not solve the problem of soundly checking effects for non-monadic code. Most code in Scala and Wyvern–let alone more conventional languages such as Java–is non-monadic, for good reasons: monads are restrictive and, for some kinds of programming, quite awkward. Programmers may be willing to use monads narrowly to get the benefits of control effects that Brachthäuser et al. support, but outside the Haskell community, it does not seem likely they would be willing to use them at a much broader scale for the purpose of descriptive effects.
(2) Our approach provides abstraction and polymorphism for descriptive effects. As discussed above, Brachthäuser et al.'s leverage of Scala's type members provides abstraction and polymorphism only for prescriptive effects, and only in the context of monadic code. Even setting aside the issue of monads, above, it is unclear how their approach can provide abstraction for descriptive effects. The reason is that their abstraction works backwards from the kind we need. For example, we want to be able to implement a logger in terms of file I/O–and hide the fact that it is implemented that way. The natural way to start would be to model file I/O in their system as a set of effect operations that "handle" I/O operations at the top level (their system does not provide support for this, so it would have to be added). A logger library could then provide a set of "log" effects and a handler for them, implemented in terms of the top-level I/O operations. But it would not be possible to hide the fact that the logger library was implemented in terms of the I/O operations, because the handler for the log effects would have to be annotated with I/O operation effects. Furthermore, all log operations would have to be nested in the scope of the log handler, annoyingly inverting control flow relative to the expected approach. And this would have to be done for every library that abstract from the built-in I/O effects, a highly anti-modular approach.
(3) Our contribution applying our effect system to security reasoning does not have an analog in Brachthäuser et al.
(4) We formalize our effect system and prove its soundness. Furthermore, we formalize authority and authority attenuation. Brachthäuser et al. contains no formalization, and does not consider authority or authority attenuation.
(5) Our implementation and case study considers side effects and security issues. Brachthäuser et al. also present an implementation and case studies, but focus on very different issues, mainly control constructs.

In summary, despite the fact that Brachthäuser et al. use Scala's type members, including their support for bounded type polymorphism and abstraction, for prescriptive, algebraic effects, this solution does not carry over to descriptive effects, the setting in which we work. Furthermore, the differences are not incidental, but foundational; it is not clear at all how to build a descriptive effect system with the expressiveness of ours using their techniques. Our contributions are thus almost entirely orthogonal to theirs.

**Coeffect systems.** Recently, coeffects have been proposed as a dual construct to effects [Petricek et al. 2014]. The relationship between effects and coeffects has been explored mostly in a denotational setting (e.g., see [Gaboardi et al. 2016]), and it is less clear how they relate in the descriptive setting

where we work. Broadly, coeffects use "input resources" to limit what a piece of code can do, whereas effects are an output that describes what it actually does. Many phenomena can be modeled either as effects or coeffects: for example, exceptions are normally viewed as an effect, but they can also be phrased as coeffects where the exception handler is the input resource. We follow both the surface-level design and language formalism from the effect literature, which is still the dominant line of research; exploring connections to coeffects is left to future work. We observe, however, that since our effects are path-dependent, they can refer to "resources" such as files that are passed in. Thus, our system includes features some kinds of expressiveness that are more typical of coeffect systems.

**Authority Attenuation.** Although a number of works on authority safety mention and explain authority attenuation (e.g., [Mettler et al. 2010; Miller 2006]), the only work on formalizing authority attenuation that we are aware of is a workshop presentation by [Loh and Drossopoulou 2017]. In the presentation, the authors used Hoare triples and invariants to show how authority can be attenuated in a restricted document object model (DOM) tree. In contrast, our approach to authority attenuation uses effect abstraction and is more general, e.g., allowing reasoning in contexts other than the DOM.

## 8 CONCLUSION

This paper presented an effect member mechanism to support effect abstraction: the ability to define higher-level effects in terms of lower-level effects, to hide that definition from clients of an abstraction, and to reveal partial information about an abstract effect through effect bounds. A set of illustrative examples as well as a framework/plugin case study demonstrates the expressiveness of effect abstraction, including the ability to support information hiding, the ability to characterize effects on application- or library-specific higher-level resources, and the ability to reason rigorously about the authority of untrusted code in a security context. We formally proved the soundness of our effect system, and showed that it is able to formally model authority attenuation for the first time. Overall, these contributions lay a solid foundation for effect systems that can scale up and can deal with the complexities of real-world code.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

## REFERENCES

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-Dependent Types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 233–249. https://doi.org/10.1145/2660193.2660216

Dor Azouri. 2018. Abusing Text Editors with Third-party Plugins. https://go.safebreach.com/rs/535-IXZ-934/images/Abusing_Text_Editors.pdf.

Dariusz Biernacki, Maciej Piròg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. In *Symposium on Principles of Programming Languages*.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *Object Oriented Programming Systems Languages and Applications*. 20. https://doi.org/10.1145/1640089.1640097

JONATHAN IMMANUEL BRACHTHÄUSER, PHILIPP SCHUSTER, and KLAUS OSTERMANN. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming* 30 (2020), e8. https://doi.org/10.1017/S0956796820000027

Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile Event Correlation with Algebraic Effects. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 67 (2018), 31 pages. https://doi.org/10.1145/3236762

Peter J. Denning. 1976. Fault Tolerant Operating Systems. *ACM Comput. Surv.* 8, 4 (Dec. 1976), 359–389. https://doi.org/10.1145/356678.356680

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming*. https://doi.org/10.1007/978-3-319-89719-6_6

Sophia Drossopoulou, James Noble, Mark S. Miller, and Toby Murray. 2016. Permission and Authority Revisited Towards a Formalisation. In *Workshop on Formal Techniques for Java-like Programs*.

Andrzej Filinski. 2010. Monads in Action. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 483–494. https://doi.org/10.1145/1706299.1706354

Jennifer Fish, Darya Melicher, and Jonathan Aldrich. 2020. A Case Study in Language-Based Security: Building an I/O Library for Wyvern. (2020), 14.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *International Conference on Functional Programming*. https://doi.org/10.1145/2951913.2951939

Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. 2013. JavaUI: Effects for Controlling UI Object Access. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–204.

Aaron Greenhouse and John Boyland. 1999. An Object-Oriented Effects System. 205–229. https://doi.org/10.1007/3-540-48743-3_10

Joseph R. Kiniry. 2006. *Advanced Topics in Exception Handling Techniques*. Springer-Verlag, Chapter Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application. http://dl.acm.org/citation.cfm?id=2124243.2124264

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In Mathematically Structured Functional Programming. https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/

K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. 2002. Using Data Groups to Specify and Check Side Effects. In *Conference on Programming Language Design and Implementation*. 12. https://doi.org/10.1145/512529.512559

Shu-Peng Loh and Sophia Drossopoulou. 2017. Specifying Attenuation. https://2017.splashcon.org/event/ocap-2017-specifying-attenuation.

Yuheng Long, Yu David Liu, and Hridesh Rajan. 2015. Intensional Effect Polymorphism. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 346–370. https://doi.org/10.4230/LIPIcs.ECOOP.2015.346

John M. Lucassen. 1987. *Types and Effects towards the Integration of Functional and Imperative Programming*. Ph.D. Dissertation. Massachusetts Institute of Technology.

John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Symposium on Principles of Programming Languages*. 11. https://doi.org/10.1145/73560.73564

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy*. 16.

Daniel Marino and Todd Millstein. 2009. A Generic Type-and-effect System. In *International Workshop on Types in Language Design and Implementation*. 12. https://doi.org/10.1145/1481861.1481868

Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *European Conference on Object-Oriented Programming*.

Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed System Security Symposium*.

Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University.

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. Program. Lang. Syst.* 10, 3 (July 1988), 470–502. https://doi.org/10.1145/44501.45065

Toby Murray. 2008. Analysing Object-Capability Security. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*.

Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. *SIGPLAN Not.* 40, 10 (Oct. 2005), 41–57. https://doi.org/10.1145/1103845.1094815

David L. Parnas. 1971. Information Distribution Aspects of Design Methodology. *IFIPS Congress* 71, 339–344.

David L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (1972), 1053–1058. https://doi.org/10.1145/361598.361623

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *International Conference on Functional Programming*. https://doi.org/10.1145/2628136.2628160

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*.

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *European Conference on Object-Oriented Programming*. 25. https://doi.org/10.1007/978-3-642-31057-7_13

Valery Trifonov and Zhong Shao. 1999. Safe and Principled Language Interoperation. In *European Symposium on Programming Languages and Systems*.

Franklyn A. Turbak and David K. Gifford. 2008. *Design Concepts in Programming Languages*. The MIT Press.

Marko van Dooren and Eric Steegmans. 2005. Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions Using Anchored Exception Declarations. *SIGPLAN Not.* 40, 10 (Oct. 2005), 455–471. https://doi.org/10.1145/1103845.1094847

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 5 (2019), 29 pages. https://doi.org/10.1145/3290318

## A PROOF OF THEOREM 2 (TRANSITIVITY OF SUBTYPING)

**Lemma 2.** *If* $\Gamma, x : \tau \vdash \varepsilon_1 <: \varepsilon_2$, *and* $\Gamma \vdash \tau' <: \tau$, *then* $\Gamma, x : \tau' \vdash \varepsilon_1 <: \varepsilon_2$.

Proof. The proof is by structural induction on the rule to derive $\Gamma, x : \tau \vdash \varepsilon_1 <: \varepsilon_2$.

(1) Subeffect-Subset
Since the premise doesn't rely on the context. This case is trivially true.

(2) Subeffect-Upperbound
If the type of $n$ is not changed, then we can apply the same rule to to derive $\Gamma, x : \tau' \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2$. If the type of $n$ is replaced by $\tau'$, then we have $\texttt{effect}\ \ g \leqslant \varepsilon' \in \sigma$, where $\Gamma, n : \tau' \vdash \varepsilon' <: \varepsilon$. By IH, we have $\Gamma, n : \tau' \vdash [n/y]\varepsilon \cup \varepsilon_1 <: \varepsilon_2$. By transitivity of subeffecting, we have $\Gamma, n : \tau' \vdash [n/y]\varepsilon' \cup \varepsilon_1 <: \varepsilon_2$. Then we can apply Subeffect-Upperbound again to derive $\Gamma, x : \tau' \vdash \varepsilon_1 \cup \{n.g\} <: \varepsilon_2$.

(3) Subeffect-Lowerbound
This case is similar to Subeffect-Upperbound

(4) Subeffect-Def-1
Since the declaration type $\texttt{effect}\ g = \{\varepsilon\}$ is not changed, the result follows directly by induction hypothesis.

(5) Subeffect-Def-2
Since the declaration type $\texttt{effect}\ g = \{\varepsilon\}$ is not changed, the result follows directly by induction hypothesis.

□

**Lemma 3.** *If* $\Gamma, x : \tau \vdash \tau_1 <: \tau_2$, *and* $\Gamma \vdash \tau' <: \tau$, *then* $\Gamma, x : \tau' \vdash \tau_1 <: \tau_2$
*If* $\Gamma, x : \tau \vdash \sigma_1 <: \sigma_2$, *and* $\Gamma \vdash \tau' <: \tau$, *then* $\Gamma, x : \tau' \vdash \sigma_1 <: \sigma_1$

Proof. We induct on the number of S-Alg used to derive the typing judgment in the premise of the statement.

BC S-Alg is not used, so we have $\Gamma, x : \tau \vdash \sigma_1 <: \sigma_2$ derived by S-Refl2 or one of the S-Effect rules. The proof is trivial if we apply lemma 2.

IS1 Assume we used S-Alg n times to derive $\Gamma, x : \tau \vdash \{y \Rightarrow \sigma_i^{i \in 1...m}\} <: \Gamma \vdash \{y \Rightarrow {\sigma'}_i^{i \in 1...n}\}$. Then for each subtyping judgments in the premise of S-Alg, we can apply induction hypothesis to derive $\Gamma,\ x : \tau',\ y : \{y \Rightarrow \sigma_i^{i \in 1..m}\} \vdash \sigma_{p(i)} <: \sigma'_i$. Then by applying S-Alg, we have $\Gamma, x : \tau' \vdash \{y \Rightarrow \sigma_i^{i \in 1...m}\} <: \Gamma \vdash \{y \Rightarrow {\sigma'}_i^{i \in 1...n}\}$

IS2 Assume we used S-Alg n times to derive $\Gamma, y : \tau \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 <: \texttt{def}\ m(x : \tau'_1) : \{\varepsilon_2\}\ \tau'_2$, by inversion on S-Def, we have $\Gamma, y : \tau \vdash \tau'_1 <: \tau_1, \Gamma, y : \tau \vdash \tau_2 <: \tau'_2$, and $\Gamma, y : \tau, x : \tau_1 \vdash \varepsilon_1 <: \varepsilon_2$. Then by induction hypothesis and lemma 2, we have $\Gamma, y : \tau' \vdash \tau'_1 <: \tau_1, \Gamma, y : \tau' \vdash \tau_2 <: \tau'_2$, and $\Gamma, y : \tau', x : \tau_1 \vdash \varepsilon_1 <: \varepsilon_2$. Then we use S-Def to derive $\Gamma, y : \tau' \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2 <: \texttt{def}\ m(x : \tau'_1) : \{\varepsilon_2\}\ \tau'_2$

□

*A.0.1 Proof of theorem 2.* If $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$, then $\Gamma \vdash \tau_1 <: \tau_3$.
If $\Gamma \vdash \sigma_1 <: \sigma_2$ and $\Gamma \vdash \sigma_2 <: \sigma_3$, then $\Gamma \vdash \sigma_1 <: \sigma_3$.

Proof. We induct on the the number of S-Alg used to derive the two judgments in the premise of the first statement: $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$, or the two judgments in the premise of the second statement: $\Gamma \vdash \sigma_1 <: \sigma_2$ and $\Gamma \vdash \sigma_2 <: \sigma_3$.

BC The S-Alg is not used, so we have $\Gamma \vdash \sigma_1 <: \sigma_2$ and $\Gamma \vdash \sigma_2 <: \sigma_3$ by S-Refl2 or one of S-Effect. By lemma 7 transitivity of subeffecting, it is easy to see $\Gamma \vdash \sigma_1 <: \sigma_3$

IS1 Assume we used S-Alg $n$ times to derive $\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1\ldots m}\} <: \{x \Rightarrow {\sigma'}_i^{i\in 1\ldots n}\}$ and $\Gamma \vdash \{x \Rightarrow {\sigma'}_i^{i\in 1\ldots n}\} <: \{x \Rightarrow {\sigma''}_i^{i\in 1\ldots k}\}$. By inversion of S-Alg, there is an injection $p : \{1..n\} \mapsto \{1..m\}$ such that $\forall i \in 1..n,\ \Gamma, x : \{x \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma_{p(i)} <: \sigma'_i$. There is another injection $q : \{1..k\} \mapsto \{1..n\}$ such that $\forall i \in 1..k,\ \Gamma, x : \{x \Rightarrow {\sigma'}_i^{i\in 1..n}\} \vdash \sigma'_{q(i)} <: \sigma''_i$. So for each $i \in 1..k$ we have two judgments

$$\Gamma, x : \{x \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma_{p(q(i))} <: \sigma'_{q(i)}$$
$$\Gamma, x : \{x \Rightarrow {\sigma'}_i^{i\in 1..n}\} \vdash \sigma'_{q(i)} <: \sigma''_i$$

By lemma 3, we can write the second judgment as $\Gamma, x : \{x \Rightarrow \sigma_i^{i\in 1..m}\} \vdash \sigma'_{q(i)} <: \sigma''_i$. By IH, for all $i \in 1..k$, $\Gamma, x : \{x \Rightarrow {\sigma''}_i^{i\in 1..k}\} \vdash \sigma_{p(q(i))} <: \sigma''_i$. Since the function $p \circ q$ is a bijection from $\{1..k\} \mapsto \{1..n\}$, we can use the rule S-Alg again to derive $\Gamma \vdash \{x \Rightarrow \sigma_i^{i\in 1\ldots m}\} <: \{x \Rightarrow {\sigma''}_i^{i\in 1\ldots k}\}$

IS2 Assume we used S-Alg $n$ times to derive $\Gamma \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau'_1 <: \texttt{def}\ m(x : \tau_2) : \{\varepsilon_2\}\ \tau'_2$ and $\Gamma \vdash \texttt{def}\ m(x : \tau_2) : \{\varepsilon_2\}\ \tau'_2 <: \texttt{def}\ m(x : \tau_3) : \{\varepsilon_3\}\ \tau'_3$. By inverse on S-Def, we have $\Gamma \vdash \tau_2 <: \tau_1$, $\Gamma \vdash \tau_3 <: \tau_2$, $\Gamma \vdash \tau'_1 <: \tau'_2$, and $\Gamma \vdash \tau'_2 <: \tau'_3$. By IH, we have $\Gamma \vdash \tau'_1 <: \tau'_3$ and $\Gamma \vdash \tau_3 <: \tau_1$. We have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$ by transitivity of subeffects. Hence we can use S-Def again to derive $\Gamma \vdash \texttt{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau'_1 <: \texttt{def}\ m(x : \tau_3) : \{\varepsilon_3\}\ \tau'_3$.

IS3 By transitivity of subeffecting, other cases for $\Gamma \vdash \sigma_1 <: \sigma_3$ are trivial.

□

# B PROOF OF THE TYPE SOUNDNESS THEOREMS

## B.1 Lemmas

Proof. Straightforward induction on typing derivations. □

**Lemma 4** (Weakening). *If* $\Gamma \mid \emptyset \vdash e : \{\varepsilon\}\ \tau$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma,\ x : \tau' \mid \emptyset \vdash e : \{\varepsilon\}\ \tau$, *and the latter derivation has the same depth as the former.*

Proof. Straightforward induction on typing derivations. □

**Lemma 5** (Reverse of Subeffecting-Lowerbound). *If* $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ , $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, *and* $\texttt{effect}\ g \leqslant \varepsilon \in \sigma$ *then* $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$

Proof. We prove this by induction on $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\})$, which is defined in Fig. 12

BC If $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\}) = 0$. Then $x.g$ can not have a definition. This case is vacuously true.

IS We case on the rule used to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$:

(a) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Subset: If $x.g \notin \varepsilon_1$, then we can use Subeffect-Subset to show $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$ If $x.g \in \varepsilon_1$. Then $\varepsilon_1 = \varepsilon'_1 \cup \{x.g\}$, where $\varepsilon'_1 \subseteq \varepsilon_2$. So we can use Subeffect-Def-1 to show $\Gamma \vdash \varepsilon'_1 \cup \{x.g\} <: \varepsilon_2 \cup [x/y]\varepsilon$

(b) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Upperbound:
Then we have $\varepsilon_1 = \varepsilon'_1 \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ h = \{\varepsilon'\} \in \sigma$, and $\Gamma \vdash \varepsilon'_1 \cup [z/y']\varepsilon' <: \varepsilon_2 \cup \{x.g\}$ By IH, we have $\Gamma \vdash \varepsilon'_1 \cup [z/y']\varepsilon' <: \varepsilon_2 \cup [x/y]\varepsilon$ Using Subeffect-Upperbound, we have $\Gamma \vdash \varepsilon'_1 \cup \{z.h\} <: \varepsilon_2 \cup [x/y]\varepsilon$

(c) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Def-1:
If Subeffect-Def-1 uses the effect $x.g$, then we immediately have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$ Otherwise, if Subeffect-Def-1 doesn't use $x.g$, then we have $\varepsilon_2 = \varepsilon'_2 \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ h = \{\varepsilon'\} \in \sigma$, and $\Gamma \vdash \varepsilon_1 <: \varepsilon'_2 \cup [z/y']\varepsilon' \cup \{x.y\}$. By IH, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon'_2 \cup [z/y']\varepsilon' \cup [x/y]\varepsilon$. Using Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup [x/y]\varepsilon$

(d) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2 \cup \{x.g\}$ is derived by Subeffect-Def-2:
This case is similar to (b)

□

**Lemma 6** (Reverse of Subeffecting-Def-2). *If* $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ , $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, *and* $\texttt{effect}\ g = \{\varepsilon\} \in \sigma$ *then* $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$

Proof. We prove this by induction on $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\})$, which is defined in Fig. 12

BC If $size(\varepsilon_1 \cup \varepsilon_2 \cup \{x.g\}) = 0$. Then $x.g$ can not have a definition. This case is vacuously true.

IS We case on the rule used to derive $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$:

(a) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Subset:
Then $x.g \in \varepsilon_2$. So we can use Subeffect-Def-1 to derive $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$

(b) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Upperbound:
If the Subeffect-Upperbound rule uses the effect $x.g$, then we by the premise of Subeffect-Upperbound, we have $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$ If the Subeffect-Upperbound rule does not use the effect $x.g$, then we have $\varepsilon_1 = \varepsilon_1' \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ h \leqslant \varepsilon' \in \sigma$, and $\Gamma \vdash \varepsilon_1' \cup [z/y']\varepsilon' \cup \{x.g\} <: \varepsilon_2$ By IH, we have $\Gamma \vdash \varepsilon_1' \cup [z/y']\varepsilon' \cup [x/y]\varepsilon <: \varepsilon_2$. Using Subeffect-Upperbound, we derive $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2$.

(c) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Def-1:
Then we have $\varepsilon_2 = \varepsilon_2' \cup \{z.h\}$, $\Gamma \vdash z : \{y' \Rightarrow \sigma\}$, $\texttt{effect}\ h = \{\varepsilon'\} \in \sigma$, and $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2' \cup [z/y']\varepsilon'$. By IH, we have $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2' \cup [z/y']\varepsilon'$. Using Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 \cup [x/y]\varepsilon <: \varepsilon_2 \cup \{z.h\}$.

(d) $\Gamma \vdash \varepsilon_1 \cup \{x.g\} <: \varepsilon_2$ is derived by Subeffect-Def-2:
This case is similar to (b)

□

**Lemma 7** (Transitivity in subeffecting). *If* $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ *and* $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$, *then* $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

Proof. We prove this using structural induction on $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3)$, which is defined in Fig. 12

BC Let $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3) = 0$. The judgments $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ are derived from Subeffect-Subset. So we have transitivity immediately.

IS Let $N \geq 0$, assume $\forall \varepsilon_1, \varepsilon_2, \varepsilon_3$ with $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3) \leq N$, if $\varepsilon_1 <: \varepsilon_2$ and $\varepsilon_2 <: \varepsilon_3$, then $\varepsilon_1 <: \varepsilon_3$. Let $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ and $size(\Gamma, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3) = N + 1$. Want to show $\varepsilon_1 <: \varepsilon_3$. We case on the rules used to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$

(a) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Subset

(i) $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Subset.
Transitivity in this case is trivial.

(ii) $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Upperbound.
Let $\varepsilon_2 = \varepsilon_2' \cup \{x.g\}$. By Subeffect-Upperbound, we have $\Gamma \vdash x : \{y \Rightarrow \sigma\}$ $\texttt{effect}\ g \leqslant \varepsilon \in \sigma$ and $\varepsilon_2' \cup [x/y]\varepsilon <: \varepsilon_3$ There are two cases:

(A) If $\{x.g\} \notin \varepsilon_1$, then $\varepsilon_1 \subseteq \varepsilon_2'$. Therefore $\Gamma \vdash \varepsilon_1 <: \varepsilon_2' \cup [x/y]\varepsilon$. By induction hypothesis, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

(B) If $\{x.g\} \in \varepsilon_1$, then $\varepsilon_1 = \varepsilon_1' \cup \{x.g\}$, and $\varepsilon_1' \subseteq \varepsilon_2'$. So we have $\Gamma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_2' \cup [x/y]\varepsilon$ by Subeffect-Subset. By IH, we have $\varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_3$. Then we use Subeffect-Upperbound to derive $\varepsilon_1' \cup \{x.g\} <: \varepsilon_3$

(iii) $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-1.
Let $\varepsilon_3 = \varepsilon_3' \cup \{x.g\}$. We have $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, $\texttt{effect}\ g = \{\varepsilon\}$, and $\Gamma \vdash \varepsilon_2 <: \varepsilon_3' \cup [x/y]\varepsilon$.

By IH, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3' \cup [x/y]\varepsilon$ Then we can use Subeffect-Def-1 again to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$

(iv) $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-2.
The proof is identical to ii.

(b) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Upperbound.
So we have $\varepsilon_1 = \varepsilon_1' \cup \{x.g\}$. $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, $\texttt{effect}\ g = \{\varepsilon\}$, and $\Gamma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_2$. Using IH, we have $\Gamma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_3$. Using Suveffect-Upperbound again, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

(c) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Def-1.
Therefore we let $\varepsilon_2 = \varepsilon_2' \cup \{x.g\}$, $\Gamma \vdash x : \{y \Rightarrow \sigma\}$, and $effect\ g = \{\varepsilon\} \in \sigma$. By premise of Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 <: [x/y]\varepsilon \cup \varepsilon_2'$. Since $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$, we have $\Gamma \vdash \varepsilon_2' \cup \{x.g\} <: \varepsilon_3$.

(i) $\Gamma \vdash \varepsilon_2' \cup \{x.g\} <: \varepsilon_3$ by Subeffect-Subset
Then we have $\varepsilon_3 = \varepsilon_3' \cup \{x.g\}$, and $\varepsilon_2' \subseteq \varepsilon_3'$. Therefore we have $\varepsilon_2' \cup [x/y]\varepsilon \subseteq \varepsilon_3' \cup [x/y]\varepsilon$. Therefore, $\Gamma \vdash \varepsilon_2' \cup [x/y]\varepsilon <: \varepsilon_3' \cup [x/y]\varepsilon$. By IH, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3' \cup [x/y]\varepsilon$. By Subeffect-Def-1 ,we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_3' \cup \{x.g\} = \varepsilon_3$

(ii) $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Upperbound
Since the effect $\{x.g\}$ is used by Subeffect-Def-1, it is not used by the rule Subeffect-Upperbound. Let $\varepsilon_2 = \varepsilon_2'' \cup \{x.g\} \cup \{z.h\}$. By Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2'' \cup [x/y]\varepsilon \cup \{z.h\}$. By Subeffect-Upperbound, we have $\Gamma \vdash z : \{y' \Rightarrow \sigma'\}$, $\texttt{effect}\ h \leqslant \varepsilon' \in \sigma'$, and $\Gamma \vdash \varepsilon_2'' \cup \{x.g\} \cup [z/y']\varepsilon' <: \varepsilon_3$. By Lemma 5 and $\Gamma \vdash \varepsilon_1 <: \varepsilon_2'' \cup [x/y]\varepsilon \cup \{z.h\}$ , we have $\Gamma \vdash \varepsilon_1 <: \varepsilon_2'' \cup [x.y]\varepsilon \cup [z/y']\varepsilon'$. By Lemma 6 and $\Gamma \vdash \varepsilon_2'' \cup \{x.g\} \cup [z/y']\varepsilon' <: \varepsilon_3$, we have $\Gamma \vdash \varepsilon_2'' \cup [x/y]\varepsilon \cup [z/y']\varepsilon' <: \varepsilon_3$. Therefore, we use IH to derive $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

(iii) $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-1
Therefore, let $\varepsilon_3 = \varepsilon_3' \cup \{z.h\}$, $\Gamma \vdash z : \{y \Rightarrow \sigma'\}$, and $effect\ h = \{\varepsilon'\} \in \sigma'$. And we have $\Gamma \vdash \varepsilon_2 <: \varepsilon_3' \cup \{z.h\}$. By premise of Subeffect-Def-1, we have $\Gamma \vdash \varepsilon_2 <: [z/y]\varepsilon' \cup \varepsilon_3'$. By IH, we have $\Gamma \vdash \varepsilon_1 <: [z/y]\varepsilon' \cup \varepsilon_3'$. Using Subeffect-Def-1, we derive that $\Gamma \vdash \varepsilon_1 <: \varepsilon_3$.

(iv) $\Gamma \vdash \varepsilon_2 <: \varepsilon_3$ by Subeffect-Def-2
This case is identical to c (ii)

(d) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Def-2
This case is identical to (b)

(e) $\Gamma \vdash \varepsilon_1 <: \varepsilon_2$ by Subeffect-Lowerbound
This case is identical to (c)

□

**Lemma 8** (Substitution in types)**.** *If* $\Gamma, z : \tau \vdash \tau_1 <: \tau_2$ *and* $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau$, *then* $\Gamma \vdash [l/z]\tau_1 <: [l/z]\tau_2$. *Furthermore, if* $\Gamma, z : \tau \vdash \sigma_1 <: \sigma_2$ *and* $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau$, *then* $\Gamma \vdash [l/z]\sigma_1 <: [l/z]\sigma_2$.

Proof. The proof is by simultaneous induction on a derivation of $\Gamma, z : \tau \vdash \tau_1 <: \tau_2$ and $\Gamma, z : \tau \vdash \sigma_1 <: \sigma_2$. For a given derivation, we proceed by cases on the final typing rule used in the derivation:

*Case S-Refl1:* $\tau_1 = \tau_2$, and the desired result is immediate.

*Case S-Trans:* By inversion on S-Trans, we get $\Gamma, z : \tau \vdash \tau_1 <: \tau_2$ and $\Gamma, z : \tau \vdash \tau_2 <: \tau_3$. By the induction hypothesis, $\Gamma \vdash [l/z]\tau_1 <: [l/z]\tau_2$ and $\Gamma \vdash [l/z]\tau_2 <: [l/z]\tau_3$. Then, by

S-Trans, $\Gamma \vdash [l/z]\tau_1 <: [l/z]\tau_3$.

*Case S-Perm:* $\tau_1 = \{x \Rightarrow \sigma_i^{i\in 1..n}\}$ and $\tau_2 = \{x \Rightarrow \sigma'^{i\in 1..n}_i\}$. Substitution preserves the permutation relations, and thus, $[l/z]\{x \Rightarrow \sigma_i^{i\in 1..n}\}$ is a permutation of $[l/z]\{x \Rightarrow \sigma'^{i\in 1..n}_i\}$. Then, by S-Perm, $\Gamma \vdash [l/z]\{x \Rightarrow \sigma_i^{i\in 1..n}\} <: [l/z]\{x \Rightarrow \sigma'^{i\in 1..n}_i\}$.

*Case S-Width:* $\tau_1 = \{x \Rightarrow \sigma_i^{i\in 1..n+k}\}$ and $\tau_2 = \{x \Rightarrow \sigma_i^{i\in 1..n}\}$, and the desired result is immediate.

*Case S-Depth:* $\tau_1 = \{x \Rightarrow \sigma_i^{i\in 1..n}\}$ and $\tau_2 = \{x \Rightarrow \sigma'^{i\in 1..n}_i\}$. By inversion on S-Depth, we get $\forall i,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i\in 1..n}\},\ z : \tau \vdash \sigma_i <: \sigma'_i$. By the induction hypothesis, $\forall i,\ \Gamma,\ x : \{x \Rightarrow \sigma_i^{i\in 1..n}\} \vdash [l/z]\sigma_i <: [l/z]\sigma'_i$. Then, by S-Depth, $\Gamma \vdash [l/z]\{x \Rightarrow \sigma_i^{i\in 1..n}\} <: [l/z]\{x \Rightarrow \sigma'^{i\in 1..n}_i\}$.

*Case S-Refl2:* $\sigma_1 = \sigma_2$, and the desired result is immediate.

*Case S-Def:* $\sigma_1 = \mathsf{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2$ and $\sigma_2 = \mathsf{def}\ m(x : \tau'_1) : \{\varepsilon_2\}\ \tau'_2$. By inversion on S-Def, we get $\Gamma,\ z : \tau \vdash \tau'_1 <: \tau_1,\ \Gamma,\ z : \tau \vdash \tau_2 <: \tau'_2,\ \Gamma, z : \tau \vdash \varepsilon_1 <: \varepsilon_2$. By the induction hypothesis, $\Gamma \vdash [l/z]\tau'_1 <: [l/z]\tau_1$ and $\Gamma \vdash [l/z]\tau_2 <: [l/z]\tau'_2$. By lemma 9, $\Gamma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$. Then, by S-Def, $\Gamma \vdash [l/z](\mathsf{def}\ m(x : \tau_1) : \{\varepsilon_1\}\ \tau_2) <: [l/z](\mathsf{def}\ m(x : \tau'_1) : \{\varepsilon_2\}\ \tau'_2)$.

*Case S-Effect:* $\sigma_1 = \mathtt{effect}\ g = \{\varepsilon\}$ and $\sigma_2 = \mathtt{effect}\ g$, and the desired result is immediate.

Thus, substituting terms in types preserves the subtyping relationship. □

**Lemma 9** (Substitution in expressions and effects). *If* $\Gamma,\ z : \tau' \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$ *and* $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$, *then* $\Gamma \mid \Sigma \vdash [l/z]e : \{[l/z]\varepsilon\}\ [l/z]\tau$.

*And if* $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_2$ *and* $\Gamma \mid \Sigma \vdash l : \{\}[l/z]\tau$, *then* $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$.

*And if* $\Gamma,\ z : \tau' \mid \Sigma \vdash d : \sigma$ *and* $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$, *then* $\Gamma \mid \Sigma \vdash [l/z]d : [l/z]\sigma$.

*Furthermore, if* $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon\ wf$, *then* $\Gamma \mid \Sigma \vdash [l/z]\varepsilon\ wf$

Proof. The proof is by simultaneous induction on a derivation of $\Gamma,\ z : \tau' \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, $\Gamma,\ z : \tau' \mid \Sigma \vdash d : \sigma$, $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_1$, and $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon wf$. For a given derivation, we proceed by cases on the final typing rule used in the derivation:

*Case T-Var:* $e = x$, and by inversion on T-Var, we get $x : \tau \in (\Gamma,\ z : \tau')$. There are two subcases to consider, depending on whether $x$ is $z$ or another variable. If $x = z$, then $[l/z]x = l$ and $\tau = \tau'$. The required result is then $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau'$, which is among the assumptions of the lemma. Otherwise, $[l/z]x = x$, and the desired result is immediate.

*Case T-New:* $e = \mathsf{new}(x \Rightarrow \overline{d})$, and by inversion on T-New, we get $\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\},\ z : \tau' \mid \Sigma \vdash d_i : \sigma_i$. By the induction hypothesis, $\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \mid \Sigma \vdash [l/z]d_i : [l/z]\sigma_i$. Then, by T-New, $\Gamma \mid \Sigma \vdash \mathsf{new}(x \Rightarrow [l/z]\overline{d}) : \{\}\ \{x \Rightarrow [l/z]\overline{\sigma}\}$, i.e., $\Gamma \mid \Sigma \vdash [l/z](\mathsf{new}(x \Rightarrow \overline{d})) : \{\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$.

*Case T-Method:* $e = e_1.m(e_2)$, and by inversion on T-Method, we get $\Gamma, z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$; $\mathsf{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma}$; $\Gamma, z : \tau' \mid \Sigma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf$; and $\Gamma, z : \tau' \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$, $\mathsf{def}\ m(y : [l/z]\tau_2) : \{[l/z]\varepsilon_3\}\ [l/z]\tau_1 \in [l/z]\overline{\sigma}$, $\Gamma \mid \Sigma \vdash [l/z]([e_1/x][e_2/y]\varepsilon_3)\ wf$, and $\Gamma \mid \Sigma \vdash [l/z]e_2 : \{[l/z]\varepsilon_2\}\ [l/z][e_1/x]\tau_2$. Then, by T-Method, $\Gamma \mid \Sigma \vdash [l/z]e_1.m([l/z]e_2) : \{[l/z]\varepsilon_1 \cup [l/z]\varepsilon_2 \cup [l/z]([e_1/x][e_2/y]\varepsilon_3)\}\ [l/z]([e_1/x][e_2/y]\tau_1)$, i.e., $\Gamma \mid \Sigma \vdash [l/z](e_1.m(e_2)) : \{[l/z](\varepsilon_1 \cup \varepsilon_2 \cup [e_1/x][e_2/y]\varepsilon_3)\}\ [l/z]([e_1/x][e_2/y]\tau_1)$.

*Case T-Field:* $e = e_1.f$, and by inversion on T-Field, we get $\Gamma, z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}$ and $\mathsf{var}\ f : \tau \in \overline{\sigma}$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$ and $\mathsf{var}\ f : [l/z]\tau \in [l/z]\overline{\sigma}$. Then, by T-Field, $\Gamma \mid \Sigma \vdash ([l/z]e_1).f : \{[l/z]\varepsilon\}\ [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](e_1.f) : \{[l/z]\varepsilon\}\ [l/z]\tau$.

*Case T-Assign:* $e = (e_1.f = e_2)$, and by inversion on T-Assign, we get $\Gamma, z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$; $\mathsf{var}\ f : \tau \in \overline{\sigma}$; and $\Gamma, z : \tau' \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ \tau$. By the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}\ [l/z]\{x \Rightarrow \overline{\sigma}\}$; $\mathsf{var}\ f : [l/z]\tau \in [l/z]\overline{\sigma}$; and $\Gamma \mid \Sigma \vdash [l/z]e_2 : \{[l/z]\varepsilon_2\}\ [l/z]\tau$. Then, by T-Assign, $\Gamma \mid \Sigma \vdash [l/z]e_1.f = [l/z]e_2 : \{[l/z]\varepsilon_1 \cup [l/z]\varepsilon_2\}\ [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](e_1.f = e_2) : \{[l/z](\varepsilon_1 \cup \varepsilon_2)\}\ [l/z]\tau$.

*Case T-Loc:* $e = l$, $[l/z]l = l$, and the desired result is immediate.

*Case T-Sub:* $e = e_1$, and by inversion on T-Sub, we get $\Gamma, z : \tau' \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\tau_1$, $\Gamma, z : \tau' \mid \Sigma \vdash \tau_1 <: \tau_2$ and $\Gamma, z : \tau' \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_2$. By induction hypothesis, we have $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_1\}[l/z]\tau_1$, $\Gamma \mid \Sigma \vdash [l/z]\tau_1 <: [l/z]\tau_2$, and $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$. Then, by T-sub, $\Gamma \mid \Sigma \vdash [l/z]e_1 : \{[l/z]\varepsilon_2\}[l/z]\tau_2$

*Case DT-Def:* By inversion, we have $\Gamma, z : \tau,\ x : \tau_1 \mid \Sigma \vdash e : \{\varepsilon'\}\ \tau_2$, $\Gamma, z : \tau,\ x : \tau_1 \mid \Sigma \vdash \varepsilon\ wf$, $\Gamma, z : \tau \mid \Sigma \vdash \varepsilon' <: \varepsilon$, By IH, we have $\Gamma,\ x : [l/z]\tau_1 \mid \Sigma \vdash [l/z]e : \{[l/z]\varepsilon'\}\ [l/z]\tau_2$, $\Gamma,\ x : [l/z]\tau_1 \mid \Sigma \vdash [l/z]\varepsilon\ wf$, $\Gamma \mid \Sigma \vdash [l/z]\varepsilon' <: [l/z]\varepsilon$. By DT-Def, we have $\Gamma \mid \Sigma \vdash \mathsf{def}\ m(x : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2 = [l/z]e : \mathsf{def}\ m(x : [l/z]\tau_1) : \{[l/z]\varepsilon\}\ [l/z]\tau_2$

*Case DT-Var:* $d = \mathsf{var}\ f : \tau = n$, and by definition of $n$, there are two subcases:

*Subcase n is x:* In this case, $d = \mathsf{var}\ f : \tau = x$, and by inversion on DT-Var, we get $\Gamma, z : \tau' \mid \Sigma \vdash x : \{\}\ \tau$. There are two subcases to consider, depending on whether $x$ is $z$ or another variable. If $x = z$, then by the induction hypothesis, $\Gamma \mid \Sigma \vdash [l/z]x : \{\}\ [l/z]\tau$, which yields $\Gamma \mid \Sigma \vdash l : \{\}\ [l/z]\tau$ and $\tau = \tau'$, and thus, $\Gamma \mid \Sigma \vdash \mathsf{var}\ f : [l/z]\tau = l : \mathsf{var}\ f : [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](\mathsf{var}\ f : \tau = l) : [l/z](\mathsf{var}\ f : \tau)$, as required. If $x \neq z$, then $\Gamma \mid \Sigma \vdash [l/z]x : \{\}\ [l/z]\tau$ yields $\Gamma \mid \Sigma \vdash x : \{\}\ [l/z]\tau$, and thus, $\Gamma \mid \Sigma \vdash \mathsf{var}\ f : [l/z]\tau = x : \mathsf{var}\ f : [l/z]\tau$, i.e., $\Gamma \mid \Sigma \vdash [l/z](\mathsf{var}\ f : \tau = x) : [l/z](\mathsf{var}\ f : \tau)$, as required.

*Subcase n is l:* In this case, $d = \mathsf{var}\ f : \tau = l$, i.e., the field is resolved to a location $l$. This is not affected by the substitution, and the desired result is immediate.

*Case DT-Effect:* By IH, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon\ wf$. We use DT-Effect to derive $\Gamma \mid \Sigma \vdash \mathtt{effect}\ g = \{[l/z]\varepsilon\} : \mathtt{effect}\ g = \{[l/z]\varepsilon\}$

*Case Subeffect-Subset:* By inversion, we have $\varepsilon_1 \subseteq \varepsilon_2$. So $[l/z]\varepsilon_1 \subseteq [l/z]\varepsilon_2$. By Subeffect-Subset, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$.

*Case Subeffect-Upperbound:* By inversion, we have $\varepsilon_1 = \varepsilon_1' \cup \{x.g\}$, $\Gamma, z : \tau \mid \Sigma \vdash x : \{y \Rightarrow \sigma\}$, $effect\ g \leqslant \{\varepsilon\} \in \sigma$ and $\Gamma, z : \tau \mid \Sigma \vdash \varepsilon_1' \cup [x/y]\varepsilon <: \varepsilon_2$. By IH, we have

$\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [l/z][x/y]\varepsilon <: [l/z]\varepsilon_2$. Since $y$ is a free variable, we select $y$ such that $x \neq y$ and $y \neq z$. We case on if $z = x$:

(1) If $z \neq x$, then we can swap the order of the substitutions on $\varepsilon$ $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [x/y][l/z]\varepsilon <: [l/z]\varepsilon_2$. Using substitution lemma for typing on $\Gamma, z : \tau \mid \Sigma \vdash x : \{y \Rightarrow \sigma\}$, we have $\Gamma \mid \Sigma \vdash x : \{y \Rightarrow [l/z]\sigma\}$, $\texttt{effect}\ g \leqslant [l/z]\varepsilon \in [l/z]\sigma$. Using Subeffect-Upperbound, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup \{x.g\} <: [l/z]\varepsilon_2$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$.

(2) If $z = x$ Then we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [l/x, y]\varepsilon <: [l/z]\varepsilon_2$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup [l/y][l/z]\varepsilon <: [l/z]\varepsilon_2$ We case on the derivation of $\Gamma, z : \tau \mid \Sigma \vdash z : \{y \Rightarrow \sigma\}$.

(a) (T-Var)

$$\frac{z : \tau \in \Gamma, z : \tau}{\Gamma, z : \tau \mid \Sigma \vdash z : \tau}$$

So $\tau = \{y \Rightarrow \sigma\}$. By our assumption, we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$. Since $\texttt{effect}\ g \leqslant \varepsilon \in \sigma$, we have $\texttt{effect}\ g \leqslant [l/z]\varepsilon \in [l/z]\sigma$. Therefore, we can use Subeffect-Upperbound on $\{l.g\}$ to derive $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1' \cup \{l.g\} <: [l/z]\varepsilon_2$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$

(b) (T-Sub)

$$\frac{\Gamma, z : \tau \mid \Sigma \vdash z : \tau_1 \quad \Gamma, z : \tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}}{\Gamma, z : \tau \mid \Sigma \vdash z : \{y \Rightarrow \sigma\}}$$

Notice that we introduced a new type $\tau_1$ that $z$ can be ascribed to. The judgment $\Gamma, z : \tau \mid \Sigma \vdash z : \tau_1$ can be derived by T-Sub, which introduce a new type $\tau_2$ such that $\Gamma, z : \tau \mid \Sigma \vdash \tau_2 <: \tau_1$, or T-Var, which shows $\tau_1 = \tau$. Therefore if we follow the derivation tree, we get a chain relation $\Gamma, z : \tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}$, $\Gamma, z : \tau \mid \Sigma \vdash \tau_2 <: \tau_1$, $\ldots$, $\Gamma, z : \tau \mid \Sigma \vdash \tau <: \tau_n$. We can apply IH on these judgments, so we have a chain $\Gamma \mid \Sigma \vdash [l/z]\tau_1 <: \{y \Rightarrow [l/z]\sigma\}$, $\Gamma \mid \Sigma \vdash [l/z]\tau_2 <: [l/z]\tau_1 \ldots$, $\Gamma \mid \Sigma \vdash [l/z]\tau <: [l/z]\tau_n$. By transitivity of subtyping, we have $\Gamma \mid \Sigma \vdash [l/z]\tau <: \{y \Rightarrow [l/z]\sigma\}$ So we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$ The rest of the proof is similar to case (a).

*Case SUBEFFECT-DEF-1:* By inversion, we have $\varepsilon_2 = \varepsilon_2' \cup \{x.g\}$, $\Gamma, z : \tau \mid \Sigma \vdash x : \{y \Rightarrow \sigma\}$, $\texttt{effect}\ g = \{\varepsilon\} \in \sigma$, and $\Gamma, z : \tau \mid \Sigma \vdash \varepsilon_1 <: \varepsilon_2' \cup [x/y]\varepsilon$. By IH, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [l/z][x/y]\varepsilon$. Since y is a free variable, we can select y such that $y \neq x$ and $y \neq z$. We case on if $x = z$:

(1) If $z \neq x$, then $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [x/y][l/z]\varepsilon$ By substitution lemma for typing, we have $\Gamma \mid \Sigma \vdash x : \{y \Rightarrow [l/z]\sigma\}$, $\texttt{effect}\ g = [l/z]\varepsilon \in [l/z]\sigma$. Using Subeffect-Def-1, we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup \{x.g\}$, which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$

(2) If $z = x$ Then we have $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [l/x, y]\varepsilon$, which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup [l/y][l/z]\varepsilon$

We case on the derivation of $\Gamma, z : \tau \mid \Sigma \vdash z : \{y \Rightarrow \sigma\}$.

(a) (T-Var)

$$\frac{z : \tau \in \Gamma, z : \tau}{\Gamma, z : \tau \mid \Sigma \vdash z : \tau}$$

So $\tau = \{y \Rightarrow \sigma\}$. By our assumption, we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$. Since $\texttt{effect}\ g = \{\varepsilon\} \in \sigma$, we have $\texttt{effect}\ g = \{[l/z]\varepsilon\} \in [l/z]\sigma$. Therefore, we can use Subeffect-Def-1 on $\{l.g\}$ to derive $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2' \cup \{l.g\}$, Which is equivalent to $\Gamma \mid \Sigma \vdash [l/z]\varepsilon_1 <: [l/z]\varepsilon_2$

(b) (T-Sub)

$$\frac{\Gamma, z : \tau \mid \Sigma \vdash z : \tau_1 \quad \Gamma, z : \tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}}{\Gamma, z : \tau \mid \Sigma \vdash z : \{y \Rightarrow \sigma\}}$$

Notice that we introduced a new type $\tau_1$ that $z$ can be ascribed to. The judgment $\Gamma, z : \tau \mid \Sigma \vdash z : \tau_1$ can be derived by T-Sub, which introduce a new type $\tau_2$ such that $\Gamma, z : \tau \mid \Sigma \vdash \tau_2 <: \tau_1$, or T-Var, which shows $\tau_1 = \tau$. Therefore if we follow the derivation tree, we get a chain relation $\Gamma, z : \tau \mid \Sigma \vdash \tau_1 <: \{y \Rightarrow \sigma\}, \Gamma, z : \tau \mid \Sigma \vdash \tau_2 <: \tau_1, \ldots, \Gamma, z : \tau \mid \Sigma \vdash \tau <: \tau_n$. We can apply IH on these judgments, so we have a chain $\Gamma \mid \Sigma \vdash [l/z]\tau_1 <: \{y \Rightarrow [l/z]\sigma\}$, $\Gamma \mid \Sigma \vdash [l/z]\tau_2 <: [l/z]\tau_1, \ldots, \Gamma \mid \Sigma \vdash [l/z]\tau <: [l/z]\tau_n$. By transitivity of subtyping, we have $\Gamma \mid \Sigma \vdash [l/z]\tau <: \{y \Rightarrow [l/z]\sigma\}$. So we have $\Gamma \mid \Sigma \vdash l : \{y \Rightarrow [l/z]\sigma\}$. The rest of the proof is similar to case (a).

*Case Subeffect-Def-2:* This case is identical to *Case Subeffect-Upperbound*
*Case Subeffect-Lowerbound:* This case is identical to *Case Subeffect-Def-1*
*Case WF-Effect:* Let $n_i.g_j \in \varepsilon$ be arbitrary. By inversion, we have $\Gamma, z : \tau \mid \Sigma \vdash n_i : \{\}\{y_i \Rightarrow \overline{\sigma_i}\}$. and the effect declaration of $g_j$ is in $\overline{\sigma_i}$. By IH, we have $\Gamma \mid \Sigma \vdash [l/z]n_i : \{\}\{y_i \Rightarrow [l/z]\overline{\sigma_i}\}$ and the effect declaration of $g_j$ is in $\overline{\sigma_i}$. So we have $[l/z]\varepsilon\ wf$ by WF-Effect.

Thus, substituting terms in a well-typed expression preserves the typing. □

### B.2 Proof of Theorem 3 (Preservation)

If $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, $\mu : \Sigma$, and $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$, then $\exists \Sigma' \supseteq \Sigma$, $\mu' : \Sigma'$, $\exists \varepsilon'$, such that $\Gamma \vdash \varepsilon' <: \varepsilon$, and $\Gamma \mid \Sigma' \vdash e' : \{\varepsilon'\}\ \tau$.

Proof. The proof is by induction on a derivation of $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$. At each step of the induction, we assume that the desired property holds for all subderivations and proceed by case analysis on the final rule in the derivation. Since we assumed $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$ and there are no evaluation rules corresponding to variables or locations, the cases when $e$ is a variable (T-Var) or a location (T-Loc) cannot arise. For the other cases, we argue as follows:

*Case T-New:* $e = \mathsf{new}(x \Rightarrow \overline{d})$, and by inversion on T-New, we get $\forall i,\ d_i \in \overline{d},\ \sigma_i \in \overline{\sigma},\ \Gamma,\ x : \{x \Rightarrow \overline{\sigma}\} \mid \Sigma \vdash d_i : \sigma_i$. The store changes from $\mu$ to $\mu' = \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}$, i.e., the new store is the old store augmented with a new mapping for the location $l$, which was not in the old store ($l \notin dom(\mu)$). From the premise of the theorem, we know that $\mu : \Sigma$, and by the induction hypothesis, all expressions of $\Gamma$ are properly allocated in $\Sigma$. Then, by T-Store, we have $\mu,\ l \mapsto \{x \Rightarrow \overline{d}\} : \Sigma,\ l : \{x \Rightarrow \overline{\sigma}\}$, which implies that $\Sigma' = \Sigma,\ l : \{x \Rightarrow \overline{\sigma}\}$. Finally, by T-Loc, $\Gamma \mid \Sigma \vdash l : \{\}\ \{x \Rightarrow \overline{\sigma}\}$, and $\varepsilon' = \varnothing = \varepsilon$. Thus, the right-hand side is well typed.

*Case T-Method:* $e = e_1.m(e_2)$, and by the definition of the evaluation relation, there are two subcases:

*Subcase E-Congruence:* In this case, either $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$ or $e_1$ is a value and $\langle e_2 \mid \mu\rangle \longrightarrow \langle e_2' \mid \mu'\rangle$. Then, the result follows from the induction hypothesis and T-Method.

*Subcase E-Method:* In this case, both $e_1$ and $e_2$ are values, namely, locations $l_1$ and $l_2$ respectively. Then, by inversion on T-Method, we get that $\Gamma \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, $\mathsf{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma}$, $\Gamma \mid \Sigma \vdash [e_1/x][e_2/y]\varepsilon_3\ wf$, $\Gamma \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$, and $\varepsilon = \varepsilon_1 \cup \varepsilon_2 \cup [e_1/x][e_2/y]\varepsilon_3$. The store $\mu$ does not change, and since T-Store has been applied throughout, the store is well typed, and thus, $\Gamma \mid \Sigma \vdash \mathsf{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2 = e : \mathsf{def}\ m(x : \tau_1) : \{\varepsilon\}\ \tau_2$. Then, by inversion on DT-Def, we know that $\Gamma, x : \tau_1 \mid \Sigma \vdash e : \{\varepsilon'\}\ \tau_2$ and $\Gamma, x : \tau_1 \mid \Sigma \vdash \varepsilon' <: \varepsilon$. Finally, by the subsumption lemma, substituting locations for variables in $e$ preserves its type, and therefore, the right-hand side is well typed.

*Case T-Field:* $e = e_1.f$, and by the definition of the evaluation relation, there are two subcases:

*Subcase E-Congruence:* In this case, $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$, and the result follows from the induction hypothesis and T-Field.

*Subcase E-Field:* In this case, $e_1$ is a value, i.e., a location $l$. Then, by inversion on T-Field, we have $\Gamma \mid \Sigma \vdash l : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}$, where $\varepsilon = \emptyset$, and $\mathsf{var}\ f : \tau \in \overline{\sigma}$. The store $\mu$ does not change, and since T-Store has been applied throughout, the store is well typed, and thus, $\Gamma \mid \Sigma \vdash \mathsf{var}\ f : \tau = l_1 : \mathsf{var}\ f : \tau$. Then, by inversion on DT-Varl, we know that $\Gamma \mid \Sigma \vdash l_1 : \{\}\ \tau$ and $\varepsilon' = \emptyset = \varepsilon$, and the right-hand side is well typed.

*Case T-Assign:* $e = (e_1.f = e_2)$, and by the definition of the evaluation relation, there are two subcases:

*Subcase E-Congruence:* In this case, either $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$ or $e_1$ is a value and $\langle e_2 \mid \mu\rangle \longrightarrow \langle e_2' \mid \mu'\rangle$. Then, the result follows from the induction hypothesis and T-Assign.

*Subcase E-Assign:* In this case, both $e_1$ and $e_2$ are values, namely locations $l_1$ and $l_2$ respectively. Then, by inversion on T-Assign, we get that $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, $\mathsf{var}\ f : \tau \in \overline{\sigma}$, $\Gamma \mid \Sigma \vdash l_2 : \{\varepsilon_2\}\ \tau$, and $\varepsilon = \varepsilon_1 = \varepsilon_2 = \emptyset$. The store changes as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}/l_1 \mapsto \{x \Rightarrow \overline{d}\}]\mu$, where $\overline{d}' = [\mathsf{var}\ f : \tau = l_2/\mathsf{var}\ f : \tau = l]\overline{d}$. However, since T-Store has been applied throughout and the substituted location has the type expected by T-Store, the new store is well typed (as well as the old store), and thus, $\Gamma \mid \Sigma \vdash \mathsf{var}\ f : \tau = l_2 : \mathsf{var}\ f : \tau$. Then, by inversion on DT-Varl, we know that $\Gamma \mid \Sigma \vdash l_2 : \{\}\ \tau$ and $\varepsilon' = \emptyset$, and the right-hand side is well typed.

*Case T-Sub:* The result follows directly from the induction hypothesis.

Thus, the program written in this language is always well typed. □

### B.3 Proof of Theorem 4 (Progress)

If $\emptyset \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$ (i.e., $e$ is a closed, well-typed expression), then either

(1) $e$ is a value (i.e., a location) or
(2) $\forall \mu$ such that $\mu : \Sigma$, $\exists e', \mu'$ such that $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$.

Proof. The proof is by induction on the derivation of $\Gamma \mid \Sigma \vdash e : \{\varepsilon\}\ \tau$, with a case analysis on the last typing rule used. The case when $e$ is a variable (T-Var) cannot occur, and the case when $e$ is a location (T-Loc) is immediate, since in that case $e$ is a value. For the other cases, we argue as follows:

*Case T-New:* $e = \mathsf{new}(x \Rightarrow \overline{d})$, and by E-New, $e$ can make a step of evaluation if the new expression is closed and there is a location available that is not in the current store $\mu$. From the premise of the theorem, we know that the expression is closed, and there are infinitely many available new locations, and therefore, $e$ indeed can take a step and become a value (i.e., a location $l$). Then, the new store $\mu'$ is $\mu, l \mapsto \{x \Rightarrow \overline{d}\}$, and all the declarations in $\overline{d}$ are mapped in the new store.

*Case T-Method:* $e = e_1.m(e_2)$, and by the induction hypothesis applied to $\Gamma \mid \Sigma \vdash e_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, either $e_1$ is a value or else it can make a step of evaluation, and, similarly, by the induction hypothesis applied to $\Gamma \mid \Sigma \vdash e_2 : \{\varepsilon_2\}\ [e_1/x]\tau_2$, either $e_2$ is a value or else it can make a step of evaluation. Then, there are two subcases:

*Subcase $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$ or $e_1$ is a value and $\langle e_2 \mid \mu\rangle \longrightarrow \langle e_2' \mid \mu'\rangle$:* If $e_1$ can take a step or if $e_1$ is a value and $e_2$ can take a step, then rule E-Congruence applies to $e$, and $e$ can take a step.

*Subcase $e_1$ and $e_2$ are values:* If both $e_1$ and $e_2$ are values, i.e., they are locations $l_1$ and $l_2$ respectively, then by inversion on T-Method, we have $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$ and $\mathsf{def}\ m(y : \tau_2) : \{\varepsilon_3\}\ \tau_1 \in \overline{\sigma}$. By inversion on T-Loc, we know that the store contains an appropriate mapping for the location $l_1$, and since T-Store has been applied throughout, the store is well typed and $l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu$ with $\mathsf{def}\ m(y : \tau_1) : \{\varepsilon_3\}\ \tau_2 = e \in \overline{d}$. Therefore, the rule E-Method applies to $e$, $e$ can take a step, and $\mu' = \mu$.

*Case T-Field:* $e = e_1.f$, and by the induction hypothesis, either $e_1$ can make a step of evaluation or it is a value. Then, there are two subcases:

*Subcase $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$:* If $e_1$ can take a step, then rule E-Congruence applies to $e$, and $e$ can take a step.

*Subcase $e_1$ is a value:* If $e_1$ is a value, i.e., a location $l$, then by inversion on T-Field, we have $\Gamma \mid \Sigma \vdash l : \{\varepsilon\}\ \{x \Rightarrow \overline{\sigma}\}$ and $\mathsf{var}\ f : \tau \in \overline{\sigma}$. By inversion on T-Loc, we know that the store contains an appropriate mapping for the location $l$, and since T-Store has been applied throughout, the store is well typed and $l \mapsto \{x \Rightarrow \overline{d}\} \in \mu$ with $\mathsf{var}\ f : \tau = l_1 \in \overline{d}$. Therefore, the rule E-Field applies to $e$, $e$ can take a step, and $\mu' = \mu$.

*Case T-Assign:* $e = (e_1.f = e_2)$, and by the induction hypothesis, either $e_1$ is a value or else it can make a step of evaluation, and likewise $e_2$. Then, there are two subcases:

*Subcase $\langle e_1 \mid \mu\rangle \longrightarrow \langle e_1' \mid \mu'\rangle$ or $e_1$ is a value and $\langle e_2 \mid \mu\rangle \longrightarrow \langle e_2' \mid \mu'\rangle$:* If $e_1$ can take a step or if $e_1$ is a value and $e_2$ can take a step, then rule E-Congruence applies to $e$, and $e$ can take a step.

*Subcase $e_1$ and $e_2$ are values:* If both $e_1$ and $e_2$ are values, i.e., they are locations $l_1$ and $l_2$ respectively, then by inversion on T-Assign, we have $\Gamma \mid \Sigma \vdash l_1 : \{\varepsilon_1\}\ \{x \Rightarrow \overline{\sigma}\}$, $\mathsf{var}\ f : \tau \in \overline{\sigma}$, and $\Gamma \mid \Sigma \vdash l_2 : \{\varepsilon_2\}\ \tau$. By inversion on T-Loc, we know that the store contains an appropriate mapping for the locations $l_1$ and $l_2$, and since T-Store has been applied throughout, the store is well typed and $l_1 \mapsto \{x \Rightarrow \overline{d}\} \in \mu$ with $\mathsf{var}\ f : \tau = l \in \overline{d}$. A new well-typed store can be created as follows: $\mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}/l_1 \mapsto \{x \Rightarrow \overline{d}\}]\mu$, where $\overline{d}' = [\mathsf{var}\ f : \tau = l_2/\mathsf{var}\ f : \tau = l]\overline{d}$. Then, the rule E-Assign applies to $e$, and $e$ can take a step.

*Case T-Sub:* The result follows directly from the induction hypothesis.

Thus, the program written in this language never gets stuck. □